

1998

Exploiting cache locality at run-time

Yong Yan

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Yan, Yong, "Exploiting cache locality at run-time" (1998). *Dissertations, Theses, and Masters Projects*. Paper 1539623938.

<https://dx.doi.org/doi:10.21220/s2-h1zs-0y44>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

EXPLOITING CACHE LOCALITY AT RUN-TIME

A Dissertation

Presented by

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements of the Degree of

Doctor of Philosophy

By

Yong Yan

1998

UMI Number: 9920312

**Copyright 1999 by
Yan, Yong**

All rights reserved.

**UMI Microform 9920312
Copyright 1999, by UMI Company. All rights reserved.**

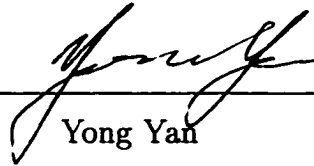
**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

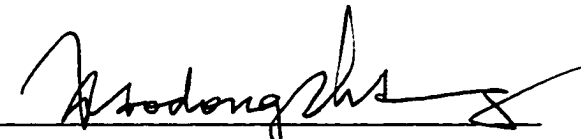
APPROVAL SHEET


This dissertation is submitted in partial fulfillment
of the requirements for the degree of

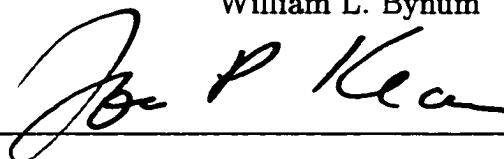
Doctor of Philosophy


Yong Yan

Approved May, 1998


Xiaodong Zhang, Dissertation Advisor


William L. Bynum

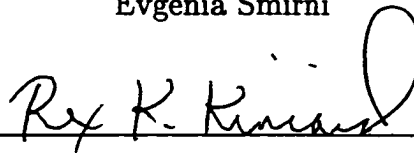

J. Phil Kearns



Rahul Simha



Evgenia Smirni



Rex K. Kincaid, Department of Mathematics

*To my parents,
my wife, Canming Jin, and
my son, Meng Yan*

Contents

Acknowledgements	viii
List of Tables	x
List of Figures	xii
Abstract	xvi
1 Introduction	2
1.1 The Problem	5
1.2 Our Approach	8
1.2.1 Estimation of The Cache-Accessing Pattern	8
1.2.2 Locality Optimizations	9
1.2.3 Trade-off Between Imbalance and Locality	9
1.2.4 Integration	9
1.2.5 Performance Evaluation	10
1.3 Contributions of This Dissertation	10
1.4 Organization of This Dissertation	11
2 Background	13
2.1 Hardware Base: The Memory Hierarchy	13

2.2	Locality Classification	15
2.3	Classification of Cache misses	17
2.4	Locality Exploitation Space	19
2.5	Related Work	24
2.5.1	User-level Approaches	24
2.5.2	Compiler Approaches	26
2.5.3	Link-time Approaches	31
2.5.4	Run-time Approaches	33
2.5.5	Operating System Based Approaches	34
2.5.6	Hardware Approaches	40
2.5.7	Summary	42
3	Cache-Locality Optimization Models	44
3.1	Programming Model	44
3.2	SMP Architecture Model	46
3.3	Cache Locality Exploitation Model	47
3.3.1	Locality Optimization of Sequential Executions	48
3.3.2	Locality Optimization of Parallel Executions	58
3.3.3	Implications of Models	59
4	System Framework and Information Abstraction	62
4.1	Run-Time System Framework	62
4.2	Information Estimation	65
4.2.1	Application Dependent Hints	65
4.2.2	Abstract Representation of Memory-Accessing Space	67
4.3	Interface and Programming Example	70
4.3.1	Application Programming Interface (API)	70

4.3.2	Classification and Programming of Applications	72
5	Memory-Layout Oriented Optimizations	80
5.1	Goals	81
5.2	Memory-access space shrinking	82
5.3	Bin Space Based Task Partitioning	86
5.3.1	Partition problem	86
5.3.2	An Algorithm to Determine Partitioning Vector	90
5.3.3	Bin Space Partitioning Procedure	95
5.4	Putting Them Together in An Implementation	99
6	Task Scheduling at Run-time	101
6.1	Overview of Existing Work and Motivation	102
6.2	An Adaptive Scheduling Algorithm	105
6.3	Variations of the Adaptive Scheduling Algorithm	110
6.4	Evaluation Methods for Scheduling Algorithms	114
6.4.1	Principles for Selecting Application Kernels	115
6.4.2	Applications	118
6.5	Experimental Results	122
6.5.1	Comparisons of Loop Scheduling Algorithms	122
6.5.2	Determine the Cost-Efficient Value	128
6.5.3	Summary of Comparisons	131
6.6	Locality-preserved Task Scheduling	132
7	Performance Evaluation	136
7.1	The Goals	136
7.2	Evaluation Method	137
7.2.1	Performance Metrics	138

7.2.2	Benchmarks	140
7.3	Performance Evaluation Environments	144
7.3.1	Event-Driven Simulator	144
7.3.2	Measurement environments	148
7.4	Performance Results	149
7.4.1	Simulation results	149
7.4.2	Measurements	160
8	Conclusions and Future Work	165
8.1	Conclusions	165
8.2	Future work	167
8.2.1	Locality Modeling	168
8.2.2	Information Estimation at Run-time	169
8.2.3	Run-time Optimizations	170
8.2.4	Programming Model Extension	171
	Bibliography	176
	Vita	189

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my advisor Dr. Xiaodong Zhang for his constant support and guidance to my research in past several years. He provides me with an incredible research environment, which allows me to have great flexibility to exploit a wide range of challenging topics from low-level architectures and communication systems to high-level performance modeling and algorithm design. My research experience with him in past five years not only has enriched my knowledge, but will also benefit my future career development.

Since I started to attack the memory bottleneck confronted by modern computer systems, I have been fortunate to have valuable advice and support from Dr. Greg Astfalk, chief scientist of scalable systems division in Hewlett Packard Corporation. Without his help, major experimental work presented in this dissertation could not have been done. No matter how busy he was, he always made time to help me with my questions. With his help, I was able to test my work on the exemplar machine. Although he is too busy to attend my dissertation defense, I consider him as an important member of the dissertation committee. I would like to express my special thanks to him here.

I also must give my special thanks to Professor Neal Wagner at the University of Texas at San Antonio. His careful proof reading and constructive suggestions on my research reports have speeded up their publications in high-quality journals and conferences.

In addition, I would like to thank all the faculty members in our department for accepting me to come to continue my research work with Dr. Xiaodong Zhang in the College of William and Mary. I would also like to thank Dr. William L. Bynum, Dr. Phil Kearns, Dr. Rahul Simha, Dr. Evgenia Smirni, and Dr. Rex K. Kincaid for serving on my dissertation committee.

I have spent more than five years in the high-performance computing and soft-

ware lab in the two Universities, where I met many talented and hardworking colleagues. I'm grateful to many of them whom I have worked with and discussed with on various research topics: Robert Castaneda, Xiaoping Chen, Sandy Dykes, Keqiang He, Canming Jin, Qian Ma, Yongshan Song, Zhichen Xu, Haixu Yang, Chenxi Zhang, and Zhao Zhang.

My research activities have been continuously supported by the National Science Foundation under grants CCR-9102854 and CCR-9400719, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, by the Office of Naval Research under grant ONR-95-1-1239.

Finally, I am indebted to my parents, my wife and my son for their love and support. They have always supported me to pursue my goal. Without them, I would not have reached my current stage.

List of Tables

4.1	Classification of applications.	72
7.1	Cache miss-rate based comparison where experiments were conducted under shrinking factor $f = 1$	149
7.2	Load balance comparison where the load balance of each program is measured by the balance coefficient defined in equation (7.1).	155
7.3	Execution performance and bus traffic: All the timing results are given in 10^6 cycles. Load balance was measured by the ratio between execution-time derivation and the mean of the execution times of multiple processors. The locality optimized programs using our run-time approach use blocking factor 1.	156
7.4	Data movement traffic: M2C, C2M, and C2C, respectively, give the total amount of data in Mega bytes moved from memory to caches, from caches to memory, and from caches to caches. The locality optimized programs using our run-time technique use blocking factor 1.	157
7.5	Effects of varying blocking factor on cache performance (2 processors were chosen for DMM_LO, and 4 processors were chosen for both AC_LO and SMM_LO): <i>misses</i> , <i>comp.</i> , and <i>rep.</i> , respectively, give the numbers of misses, compulsory misses, and replacement misses in 10^3 ; and <i>inv.</i> gives the total number of invalidations.	158

7.6	Run-time overhead in percentage of total execution time.	160
7.7	Execution time (in seconds) based comparison on HP/Convex S-class: Columns <code>time</code> and <code>overhead</code> , respectively, give total execution time and task organization overhead in second. <code>Balance</code> presents load balance mea- surements which is defined in equation (7.1). ($f = 1$).	161
7.8	The effect of different values of f on execution time (in seconds) for DMM_LO and AC_LO on four processors of HP/Convex S-class.	162
7.9	Execution time (in seconds) based comparison on HyperSPARC station- 20: Columns <code>time</code> and <code>overhead</code> , respectively, give total execution time and task organization overhead in second. <code>Balance</code> presents load balance measurements which is defined in equation (7.1). ($f = 1$).	163
7.10	The effect of different values of f on execution time (in seconds) for DMM_LO and AC_LO on four processors of HyperSPARC station-20. . .	164

List of Figures

1.1	SMP shared memory system model.	5
2.1	Pyramid views of memory hierarchies from one processor in uniprocessor systems, CC-UMA systems, and CC-NUMA systems.	14
2.2	Locality classification.	15
2.3	Cache miss classification.	17
2.4	Program execution flowchart.	21
3.1	Data-independent nested loop.	45
3.2	Addressing in an k -way set associative cache where the number of sets is 2^r , block size is 2^d , and $k = 2^c$	48
3.3	An example for the calculation of <code>postd</code> and <code>pred</code> for each memory access address in an address sequence.	51
3.4	Exemplifying Lemma 1 for a direct-mapped two-set cache with a block size of 2.	53
3.5	Exemplifying Lemma 2.	55
4.1	Execution framework of the run-time system.	64
4.2	Abstracting loop instances as tasks.	68
4.3	An abstract representation.	69

4.4	Dense matrix multiplication: the sequential program is given on the left and the locality optimized version on the run-time system is given on the right.	74
4.5	Regular computation pattern in the DMM.	75
4.6	Adjoint Convolution (AC): the sequential program is given on the left and the parallelized version on the Cacheminer system is given on the right. .	76
4.7	Irregular computation pattern in the AC.	77
4.8	Sparse matrix-matrix multiplication: the left side is the sequential program version and the right side is the rewritten version on the run-time system.	78
4.9	The dense representation in the SMM.	79
5.1	Memory-access space shrinking.	85
5.2	Partitioning patterns of an 2-dimensional bin space on four processors. .	90
5.3	Indexing partitions: the bin space is evenly divided into 4 partitions from X and Y dimensions.	98
5.4	Flowchart of task reorganization in the run-time system.	100
6.1	Performance of SOR on the KSR-1 (left figure) and Exemplar (right figure).	122
6.2	Performance of JI on the KSR-1 (left figure) and the Exemplar (right figure).	124
6.3	Performance of TC with random input on the KSR-1 (left figure) and the Exemplar (right figure).	125
6.4	Performance of TC with skewed input on the KSR-1 (left figure) and the Exemplar (right figure).	126
6.5	Performance of MM on the KSR-1 (left figure) and the Exemplar (right figure).	127
6.6	Performance of AC on the KSR-1 (left figure) and the Exemplar (right figure).	128

6.7	Performance of SOR on the KSR-1: (a) using EA with different α values (left figure); (b) using CA with different α values (right figure)	130
6.8	Performance of JI on the KSR-1: (a) using EA with different α values (left figure); (b) using CA with different α values (right figure)	131
6.9	Scheduling framework.	133
7.1	The cache-access pattern of a simply execution sequence on a processor where data a and b are mapped into the same cache block. $r(a)$ expresses a read on data a.	139
7.2	A well-tuned parallel version of the DMM application [82]. Here, pid is a thread id of value from 0 to p-1. p is the number of threads.	140
7.3	Optimizing locality and balance of AC.	141
7.4	A well-tuned parallel version of the AC application.	143
7.5	The architecture and the interfaces of the MINT simulator and the memory-hierarchy simulator.	145
7.6	Cache performance comparison between DMM_WL and DMM_LO.	150
7.7	Cache performance comparison between AC_WL and AC_LO.	151
7.8	Cache performance comparison between SMM_A and SMM_LO.	152
7.9	Execution time comparison between DMM_WL and DMM_LO.	153
7.10	Execution time comparison between AC_WL and AC_LO.	153
7.11	Execution time comparison between SMM_WL and SMM_LO.	154
8.1	An noncontiguous access pattern of a task in an array where the array is linearly laid out in memory.	169
8.2	A Type 4 benchmark: Sparse Triangular Solver (STS). Here equation $B = A \times X$ is solved where A is a sparse lower triangular matrix with a dense representation.	171
8.3	A framework of a run-time optimization system for all types of applications.	172

8.4 A generated wave-front analysis program of the STS [64]. 173

8.5 An example for the wave-front analysis. 174

ABSTRACT

With the increasing gap between the speeds of the processor and memory system, memory access has become a major performance bottleneck in modern computer systems. Recently, Symmetric Multi-Processor (SMP) systems have emerged as a major class of high-performance platforms. On these SMP systems, the efficiency of memory access in an application is critical to its overall execution performance.

Optimizing the cache locality of a parallel application is an effective approach to reduce the memory bottleneck effect to improve the performance of a parallel computation. For applications with static memory-access patterns, many effective techniques, such as compile-time locality optimizations, have been proposed. However, improving the memory performance of applications with dynamic memory-access patterns is still a hard problem in the parallel computing area. The solution to this problem is critical to the success of parallel computing because dynamic memory-access patterns occur in many real-world applications.

This dissertation is aimed at solving the above problem. Based on a rigorous analysis of cache-locality optimization, we propose a memory-layout oriented run-time technique to exploit the cache locality of parallel loops on SMP systems. The proposed technique consists of four components: (1) a method to estimate and abstract memory-access patterns of applications, (2) a memory-layout based method to reorganize tasks to maximize data reuse in caches, (3) a heuristic task partitioning algorithm to minimize both data-sharing and load imbalance, and (4) an adaptive and locality-preserved scheduling algorithm to minimize the parallel execution time of an application. These system schemes have been integrated and implemented in a run-time system.

In order to provide an insightful analysis of our run-time system, a detailed SMP simulator was built. Using simulation and measurement, we have shown our run-time approach can achieve comparable performance with compiler optimizations for those reg-

ular applications, whose load balance and cache locality can be well optimized by tiling and other program transformations. However, our approach was shown to improve significantly the memory performance for applications with dynamic memory-access patterns. Such applications are usually hard to optimize with static compiler optimizations.

The major contributions of this dissertation are:

1. We present models for the cache locality optimization problems in uniprocessor systems and SMP systems. These models characterize the complexity and present a solution framework for optimizing cache locality.
2. We present an effective internal representation for the memory-access pattern of a parallel loop to support efficient locality optimizations and information integration.
3. We present a memory-layout oriented run-time technique for locality optimization.
4. We present efficient scheduling algorithms to trade off locality and load imbalance.
5. We provide a detailed performance evaluation of the run-time optimization technique at the architecture level and at the execution level.

EXPLOITING CACHE LOCALITY AT RUN-TIME

Chapter 1

Introduction

The recent developments in circuit design, fabrication technology and Instruction-Level Parallelism (ILP) technology have increased microprocessor speed about 100% every year. However, memory-access speed has only improved about 20% every year [19]. In a modern computer system, the widening gap between processor performance and memory performance has become a major bottleneck to improving overall computer performance. Usually, a faster processor has a *higher memory-access rate*. Since the increase in memory-access speed cannot match that of the processor speed, memory-access contention is increased, which results in a longer memory-access latency. This makes memory-access operations much more expensive than computation operations. In multiprocessor systems, the effect of the widening processor-memory speed gap on performance becomes more significant due to the heavier access contention on the network and the shared memory and to the cache coherence cost. Recently, Symmetric Multi-Processor (SMP) systems have emerged as a major class of parallel computing platforms, such as HP/Convex Exemplar S-class [7], Sun SPARCcenter 2000 [14], SGI Challenge [24], and DEC AlphaServer [66]. SMPs dominate the server market for commercial applications and are used as desktops for scientific computing [67]. They are also important

building blocks for large-scale systems. The improvement on the memory performance of applications is critical to the successful use of SMP systems in the real world.

In order to narrow the processor-memory speed gap, hardware caches have been widely used to build a memory hierarchy in all kinds of computers, from supercomputers to personal computers. In addition, in a Cache-Coherent Non-Uniform Memory Access (CC-NUMA) system, the shared memory is further distributed to reduce its access-contention. The effectiveness of the memory hierarchy for improving performance of programs comes from the locality property of both instruction executions and data accesses of programs. In a short period of time, the execution of a program tends to stay in a set of instructions close in time or close in the allocation space of a program, called the *instruction locality*. Similarly, the set of instructions executed tend to access data that are also close in time or in the allocation space, called the *data locality*. Using a fast and small cache close to a CPU is expected to hold the working set of a program so that low-level memory accesses can be avoided or reduced.

Unfortunately, the memory hierarchy is not a panacea for eliminating the processor-memory performance gap. Low-level memory accesses are still substantial for many applications and are becoming more expensive as the processor-memory performance gap continues to widen. The reasons for possible slow memory accesses are:

- Applications may not be programmed with an awareness of the memory hierarchy.
- Applications have a wide range of working sets which cannot be held by a hardware cache, resulting in capacity misses at the top levels of the memory hierarchy.
- The irregular data-access patterns of applications result in excessive conflict misses at the top levels of the memory hierarchy.
- In a time-sharing system, the dynamic interaction among concurrent processes and the underlying operating system causes a considerable amount of low-level memory

accesses as processes are switched in context. This effect cannot be handled by the memory hierarchy on its own.

- In a cache coherent multiprocessor system, false data sharing and true data sharing result in considerable cache coherent misses.
- In a CC-NUMA system, processes may not be perfectly co-located with their data, which results in remote memory accesses to significantly degrade overall performance.

Due to the increasing cost of low-level memory accesses, techniques for eliminating the effect of long memory latency have been intensively investigated by researchers from application designers to hardware architects. So far, the proposed techniques fall into two categories: *latency avoidance* and *latency tolerance* [30]. The latency tolerance techniques [19] are aimed at hiding the effect of memory-access latencies by overlapping computations with communications or by aggregating communications. Most of these techniques, while reducing the impact of contentionless access latencies, do so at the cost of increasing a program's bandwidth requirements [13]. These latency tolerance techniques may increase a processor's memory bandwidth needs by causing the processor to request the same stream of operands in less time, or by causing the processor to request more data from memory. In turn, these techniques may cause the processor to stall due to queueing in the memory system. In a SMP system, it is hard for a latency tolerance technique to reduce cache-coherence overhead.

The latency avoidance techniques, also called locality optimization techniques, are aimed at minimizing low-level memory accesses by using software and hardware approaches to maximize the reusability of data or instructions at the top levels of the memory hierarchy. In a SMP system, reducing the total number of accesses at low levels of the memory hierarchy is a substantial solution to reduce cache coherence overhead, memory contention and network contention. So, the locality optimization techniques,

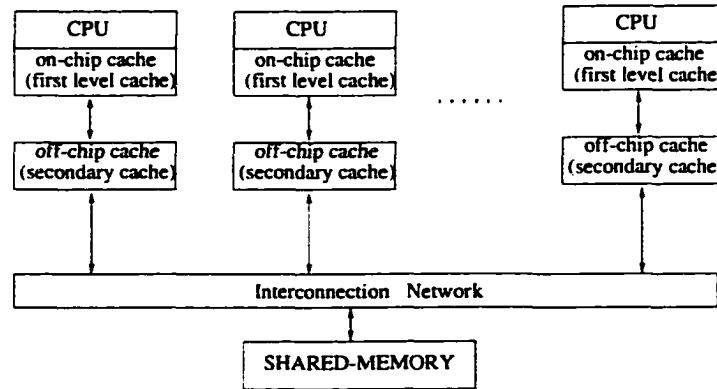


Figure 1.1: SMP shared memory system model.

i.e. the latency avoidance techniques, are more demanding than the latency tolerance techniques. In addition, because instruction accesses are more regular than data accesses, designing novel data-locality optimization techniques is more challenging and more important for performance improvement. *The objective of this dissertation is to propose an efficient technique to optimize the data cache locality of parallel applications on SMP systems.*

1.1 The Problem

In a SMP system as shown in Figure 1.1, each processor has a hierarchy of local caches (such as the on-chip cache and the off-chip cache in the figure) and all the processors share a global memory. When a processor accesses its data, it first looks up the cache hierarchy. If the data is not found in the caches, an event called as a *cache miss*, the processor reads the memory block that contains the required data from shared memory and brings a copy of the memory block in an appropriate cache block in the cache hierarchy. Data is copied into the cache hierarchy so that the subsequent accesses to the data can be satisfied from the cache and memory accesses can be avoided. The cache locality optimization is aimed at optimizing the cache-access pattern of an application so that memory accesses can be

satisfied in the cache as often as possible (or in other words, cache data can be reused as much as possible). To increase the chance of cache data to be reused, we must reduce the interference that would kick out or invalidate the cache data. In a SMP system, there are two types of interference that would affect the reuse of cache data: the interference from the local processor which refills a cache block with new data, and the interference from a remote processor which invalidates stale data copies to maintain data consistency. The two types of interference in a parallel computation are determined by how the data is accessed in the computation, called the *data-access pattern*, and how the data is mapped into a cache, called the *cache-mapping pattern*. Hence, it is essential for a cache locality optimization technique to obtain and use the information on the data-access pattern and the cache-mapping pattern of a parallel program.

The data-access pattern of a program is determined by program characteristics. Because the compilation time of an application is not a part of its execution time, a compiler can use sophisticated techniques to analyze the data-access pattern of a program. However, there is a large class of real world applications whose data-access patterns cannot be analyzed at compile-time. The data-access patterns of many these functions are dependent on run-time data. In addition, many real-world applications have indirect data accesses [81], which are difficult for a compiler to analyze. For example, pointers may point to different objects during the execution of a program, and the subscripts of an array variable may be given by another array variable. The existence of these complicated applications recommends run-time techniques for analyzing the data-access patterns.

Next, the cache-mapping pattern of a program is determined by architectural characteristics of the cache and data layout. In current commercial computer systems, caches fall into two types based on their indexing schemes: physically indexed caches, called *physical caches*, and virtually indexed caches, called *virtual caches*. In virtual caches, the mapping of data onto cache blocks is based on the virtual addresses of the

data, which is not changed by the operating system. For the complicated programs whose data-access patterns are determined by run-time data, the virtual addresses of the run-time data can only be determined at run-time. In physical caches, the mapping of data onto cache blocks is based on the physical addresses of the data in memory which are determined by the operating system at run-time. Hence, from the standpoint of analyzing cache-mapping patterns of programs, run-time analysis techniques are important for improving the memory performance of complicated applications on all cache architectures.

Along with the run-time analysis on the cache-access patterns of applications, efficient run-time locality optimizations must be applied. Because the execution time associated with the run-time analysis and optimizations, called the *run-time overhead*, extends the total execution time of an application, the design of run-time techniques is strictly constrained by the overhead. In general, the more information a run-time technique exploits, the greater number of memory accesses it can reduce, but the more run-time overhead it may cause. The challenge is how to trade off between the optimization quality and the run-time overhead. Although a compiler cannot conduct any run-time analysis, it has the advantage of being able to perform very complicated static analyses. So, an effective run-time technique should have some way to make use of the compile-time information.

In addition, because the ultimate goal of optimizing the cache locality of a parallel program is to minimize parallel computing time, locality optimizations must be carefully traded off with the other performance factors. As we have shown in our previous work [93, 94], load imbalance is an important performance factor for parallel computing. Optimizing locality and balancing load are two conflicting goals. Load balancing tends to split a group of tasks with cache affinity onto different processors, while locality optimizing tends to put tasks with cache locality affinity onto a processor. How they should be traded off is important.

The locality optimization has been a hot research topic for several years. Many effective techniques have been proposed at different system levels (more comprehensive analysis on them will be given in the next chapter). The techniques proposed at the user level mainly depend on users' analysis, which are not acceptable for general usage. The compiler-based techniques cannot handle the complicated applications (as mentioned above). At the run-time system level, locality optimization has been considered in some task scheduling systems. However, current optimizations only exploit a type of weak locality information: processor affinity, which has a very restricted application. At the operating system level and the hardware level, some locality optimization techniques are also proposed. These techniques are for improving system-wide memory performance, not a specific computation, because it is hard to get the information on the cache-access pattern of an application at the low system levels. Hence, in order to achieve efficient parallel computing for a wide range of real-world applications, conducting run-time locality optimization is essential. Run-time locality optimization can complement compiler-based techniques to handle complicated real-world applications.

1.2 Our Approach

1.2.1 Estimation of The Cache-Accessing Pattern

In order to take use of the static information of an application, we design a set of simple run-time functions which are inserted into an application by a compiler or a user to produce some application-dependent *static* information, called *hints*, into the object code. During run-time certain functions are invoked to analyze the memory-access pattern of an application based on both static and run-time information.

The memory-access pattern is abstracted in a multi-dimensional space and tasks are abstracted as points in the space. These abstractions provide a fundamental software

structure for implementation and locality optimization.

1.2.2 Locality Optimizations

Based on the predicted memory-access pattern and the architecture of the cache, two types of space transformations are conducted in the multi-dimensional memory-access space: space shrinking and space partitioning. In space shrinking, tasks are grouped based on their data affinity with the aim of maximizing cache data reuse in mind. In space partitioning, tasks are partitioned with the aim of minimizing both data sharing and load imbalance.

1.2.3 Trade-off Between Imbalance and Locality

In order to guarantee balanced execution, we propose a run-time task scheduling algorithm to trade off load imbalance and locality. If the partitions generated from locality optimization phase are well balanced, the scheduling overhead is insignificant in the execution phase, and the run-time scheduler can achieve similar efficiency of a static scheduler. The more imbalanced the partitions, the more scheduling overhead the algorithm may cause.

1.2.4 Integration

In order to minimize the run-time overhead of locality optimization, a multi-dimensional hash structure is internally built at initialization time based on application-dependent hints. Meanwhile, a compound hash function is constructed from several transformation functions, which maps M tasks into appropriate affinity groups on appropriate processors in $O(M)$ complexity. This provides an efficient integration of the task grouping and the task partitioning.

1.2.5 Performance Evaluation

We evaluated the effectiveness of our run-time technique using simulation and the measurement on commercial SMP systems. We built a detailed simulator for SMP systems to study the execution performance of our run-time system and the effect of run-time optimizations on the cache-access patterns of applications. Measurements on two SMP multiprocessors were conducted to further confirm the simulation results. The performance evaluation was conducted using several benchmarks with different program characteristics.

1.3 Contributions of This Dissertation

The primary contributions of this dissertation are as follows:

- We develop models for cache locality optimization problems in uniprocessor systems and SMP systems. These models characterize the complexities of and present a solution framework for optimizing cache locality in uniprocessor systems and in SMP systems.
- We present an effective internal representation of memory-access patterns of parallel tasks. This representation allows an efficient implementation of comprehensive locality optimizations and an efficient integration of both static information and run-time information.
- We present a memory-layout oriented run-time technique for locality optimization that contains two optimization algorithms: a task reordering algorithm and a task partitioning algorithm.
- We present an adaptive scheduling algorithm and several variations for the general scheduling problem in shared-memory systems. Then, we present a locality-

preserved scheduling algorithm to trade off locality and load imbalance.

- We provide a detailed performance evaluation of the run-time optimization technique at both the architecture level and the execution level.

1.4 Organization of This Dissertation

This dissertation contains eight chapters. In the next chapter, we introduce background knowledge about locality optimization. Then we provide a comprehensive overview of related work in order to motivate interested readers to investigate this challenging problem further.

In Chapter 3, the programming model and the architectural model are presented. Then the cache locality optimization problem and solution framework on uniprocessors and multiprocessors are described. In particular, the difficulties and complexities of optimizing the cache locality of applications on uniprocessors and multiprocessors are analyzed.

Chapter 4 presents the design principles and framework of our run-time system. The estimation technique and the internal representation of memory-access patterns are described. Furthermore, the functionality of the run-time system is described and is motivated by programming examples of several benchmarks that represent different types of applications.

Chapter 5 describes a run-time technique for optimizing the cache locality of applications. The technique consists of two components: a memory-layout based task reordering algorithm and a memory-layout based task partitioning algorithm. Because the locality-optimization oriented task partitioning is a complicated problem, this chapter describes it in detail and then presents a heuristic solution.

In Chapter 6, a general task scheduling problem is studied. Motivated by the

limits of existing scheduling techniques, an adaptive scheduling algorithm and its several variations are presented. These algorithms were experimentally evaluated and compared with existing algorithms. Then, a locality-preserved scheduling algorithm is presented to trade off load imbalance and locality in our run-time system.

Chapter 7 describes performance evaluation methods, environments, and performance evaluation results. In Chapter 8, we summarize the work presented in this dissertation and point out several future research directions. Some open questions are also discussed in this chapter.

Chapter 2

Background

Locality optimization is a complicated problem because the locality of a program is affected by a wide range of factors from the programming style at the user level to the cache architecture at the hardware-level. This chapter first gives background knowledge on the hardware base, the locality, the cache miss measure and the exploitation space of the locality optimization problem. Then, a comprehensive overview of the related work is given.

2.1 Hardware Base: The Memory Hierarchy

Building a memory hierarchy is a hardware approach widely used to bridge the processor-memory performance gap in modern computers. Locality exploitation techniques are aimed at optimizing the performance of the memory hierarchy.

From the standpoint of one processor, the memory hierarchy in a uniprocessor system or in a shared memory system has a pyramid shape in capacities and access latencies. The memory modules closer to the CPU have smaller capacity and shorter access latency than those further from the CPU. In a uniprocessor system or a Cache

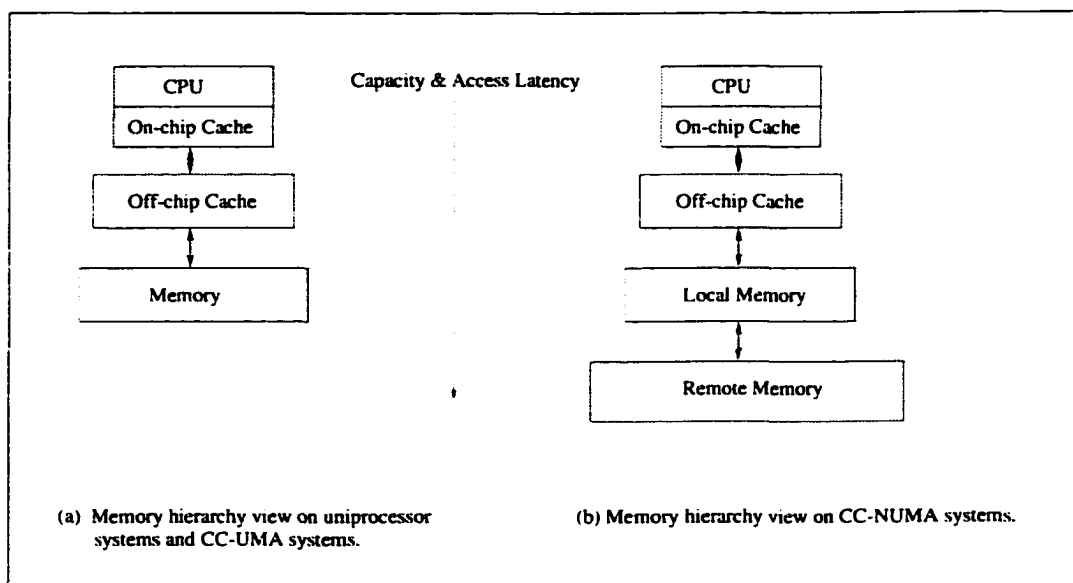


Figure 2.1: Pyramid views of memory hierarchies from one processor in uniprocessor systems, CC-UMA systems, and CC-NUMA systems.

Coherent Uniform Memory Access (CC-UMA) system, the memory hierarchy usually has three levels as shown in Figure 2.1 (a): an on-chip cache with a typical size of 8 to 64 KBytes, an off-chip cache (or secondary cache) with a typical size of 256 KBytes to 1 MByte, and a main memory that is shared and has the same access latency from multiple processors in a CC-UMA system. Some systems have a simpler memory hierarchy of one cache level and one memory level. In a CC-NUMA system, the shared main memory is distributed, where each processor has a local memory module. This structure could reduce memory-access contention and scale to a larger number of processors, resulting in a four-level memory hierarchy as illustrated in Figure 2.1(b): an on-chip cache, an off-chip cache, a local memory, and a remote memory comprised of all the local memory modules of the other processors.

In a modern computer system, hierarchy-access latency ratios are, respectively, about 3 between an off-chip cache and an on-chip cache, 15 to 30 between a local memory

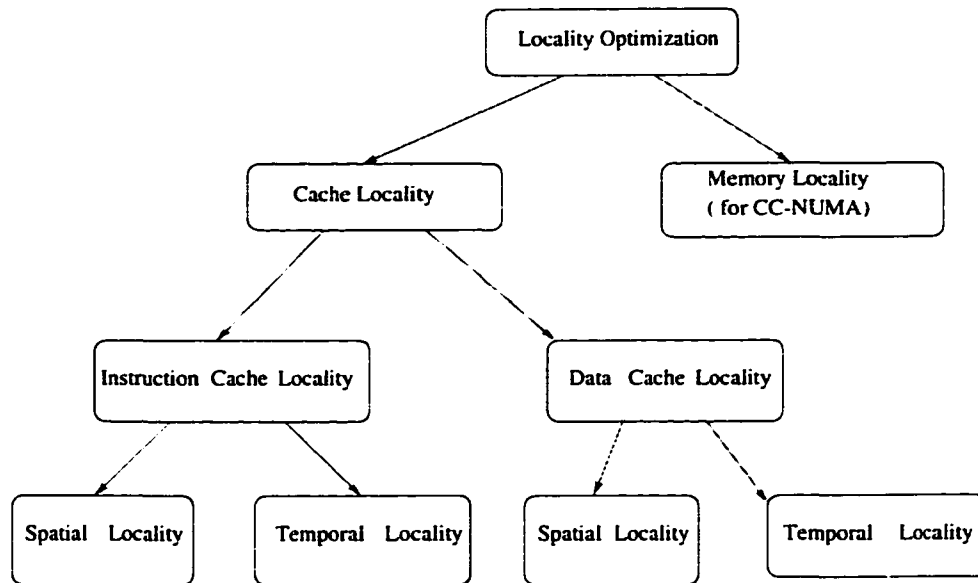


Figure 2.2: Locality classification.

and an off-chip cache, and about 3 between a remote memory module and a local memory module. The use of a memory hierarchy is aimed at taking advantage of the locality feature of the instruction executions and data accesses of a program, so that the working sets of applications will be held closer to the CPU to avoid low-level memory accesses. Top-level locality optimizations are more important than low-level locality optimizations because they will gain more improvement in overall performance.

2.2 Locality Classification

Optimizing the memory performance of programs is a major goal of locality exploitation. In general, different locality exploitation methods try to fulfill this goal by addressing different types of localities. A complete classification of localities is important for analyzing locality optimization techniques.

Programs have two distinguished types of locality properties: *temporal locality*

and *spatial locality*. The *temporal locality* refers to an observed property of most programs, i.e., once a data or instruction is referenced, it will tend to be referenced again soon. The *spatial locality* refers to the probability that once an item is referenced, nearby items will tend to be referenced soon [30]. In the memory hierarchy described in Figure 2.1, caches and memory modules are managed to exploit program locality in different approaches: the former uses a hardware-based approach, and the latter uses an operating system-based approach. Hence, with respect to the memory hierarchy, the goal of locality exploitation can be divided into two subgoals: cache locality exploitation and memory locality exploitation. (The latter is required only for CC-NUMA systems.)

Memory locality exploitation is aimed at minimizing the number of remote memory accesses by co-locating processes with their data. This process is constrained by other performance factors, such as load balance, and is mainly managed by operating system approaches. In contrast, cache-locality exploitation is aimed at minimizing the number of cache misses, and thus is more complicated than memory locality exploitation. A program execution comprises two major and different activities: instruction execution and data access. Instruction execution activities have been shown to have better locality than data access activities[30]. Most modern computers use an on-chip instruction cache and an on-chip data cache separately to exploit instruction locality and data locality. The differences between instruction execution and data access result in another separation of cache locality exploitation: instruction-locality exploitation and data-locality exploitation. A method to optimize both is ideal, but is usually difficult to design.

Regarding the two types of program locality properties, a hardware cache exploits temporal locality by placing a referenced word into the cache, and exploits spatial locality by using a cache block size larger than one word, which brings adjacent words into the cache at the same time. Optimizing the spatial locality and temporal locality of programs are two concrete subgoals of the cache-locality optimization. Based on the above analysis, locality can be further broken down into different types as shown in

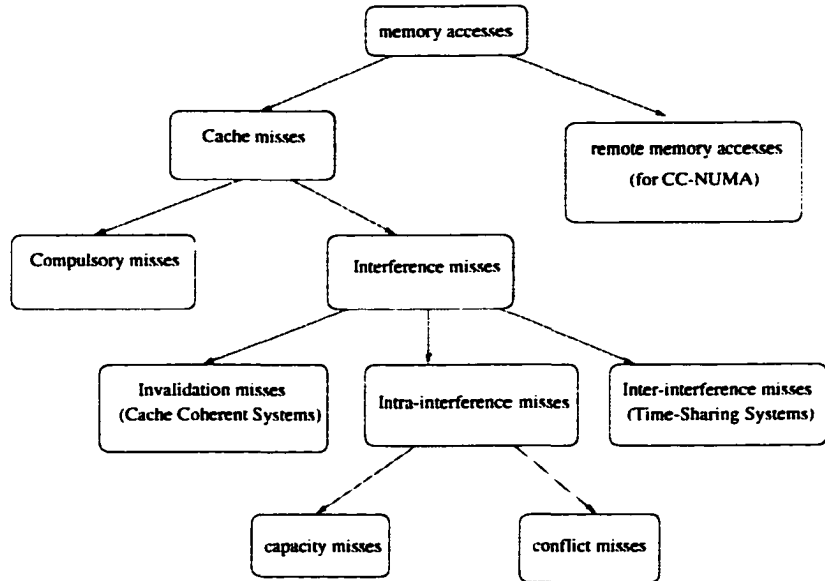


Figure 2.3: Cache miss classification.

Figure 2.2.

2.3 Classification of Cache misses

How well a locality exploitation method achieves its goal of optimizing memory performance is often quantitatively measured by the reduction in low-level memory accesses. First, memory accesses can be classified into two types: *cache misses* and *remote memory accesses* (only for CC-NUMA). Furthermore, different cache locality methods eliminate different sources that cause cache misses, resulting in different improvements on each type of cache misses. The classification of cache misses is important for evaluating locality optimization techniques.

Because the systems considered here cover uniprocessor systems and shared memory systems, which may be dedicated or time-shared, we classify cache misses into the following types:

1. *Compulsory misses*, caused by referencing an item that has never been brought into the cache before.
2. *Interference misses*, caused by referencing an item that was brought into the cache but evicted at a later time. The interference misses are further divided into the following types:
 - (a) *Intra-interference misses*, where the item to be referenced was replaced by an item of the same program at a later time.
 - (b) *Inter-interference misses*, where the item to be referenced was replaced by an item of another program at a later time.
 - (c) *Invalidation misses*, where the item to be referenced was invalidated at a later time.

The inter-interference misses come from the interference among multiple time-sharing processes. The invalidation misses come from the effect of the cache coherence protocol in a cache coherent shared memory system. The intra-interference misses come from the interference among different data objects and code segments of a program. Based on [31], the intra-interference misses can be further classified into *capacity misses*, which occur in a fully-associative cache with LRU replacement, and *conflict misses*, which occur in an n -way set associative cache, but not in a fully-associative cache. Capacity misses are caused by referencing more cache blocks than that a cache can provide. Conflict misses are caused by the mapping of multiple memory blocks into the same cache line even though empty cache lines are available. Moreover, conflict misses actually can be further classified into subtypes, such as *self-conflict misses*, which happen among the elements of an array, and *cross-conflict misses*, which happen among the elements of different arrays [17].

Based on the above decomposition of cache misses, the classification of miss types is shown in Figure 2.3.

2.4 Locality Exploitation Space

To investigate the exploitation space of locality, we should have a clear picture of the effects of different software and hardware components on the program locality during its execution. From the standpoint of one processor, the execution of a program on a uniprocessor, a CC-UMA, or a CC-NUMA has the same flowchart as illustrated in Figure 2.4. From an application program point of view, Figure 2.4 only presents the execution flowchart of a program on a uniprocessor. For a shared memory system, a program may span onto multiple processor nodes to execute. A CC-UMA processor node consists of a CPU and a cache hierarchy as described in Figure 2.4, where all processor nodes are symmetric to the shared memory. A CC-NUMA processor node comprises a CPU, a cache hierarchy, and a memory module as described in Figure 2.4, where each processor has a local memory shared by the other processors as a remote module.

Figure 2.4 shows that the lifetime cycle of a program consists of the following phases:

1. *Programming phase:* Using a programming language, a user explicitly expresses data, computation and execution control of an application in a program. The program is constructed in a virtual space. The selections of data structures, the data-access method, and execution order of computation have significant effects on the program locality. For explicit parallel programming, the partitioning methods of data layout and computation sequences are crucial to the memory performance of the program.
2. *Compilation phase:* A program is compiled to produce an object file. Advanced

compilers try to optimize all aspects of a program, from data layout to program structures. In a shared memory system, a compiler may automatically conduct a series of transformations to reorganize the data layout and computation sequence. A compilation system usually has a good chance to flexibly optimize the locality of a program because it can obtain detailed information about the program, and its compiling time is not considered significant. However, it is hard for a compiler to predict what will happen at run-time, especially with programs that have dynamic program structures and dynamic data-access patterns.

3. *Link phase*: A linker is responsible for linking the object files generated by the compiler and the object files in standard libraries and/or run-time libraries. Each object file has an independent virtual space. The linker integrates the multiple virtual spaces of the object files into a global virtual space to generate an executable file. Linking is the last phase to change the virtual space of a program.
4. *System execution phase*: The executable files generated by the linker are loaded into memory by the operating system. The virtual space of an executable file is mapped onto the physical space of memory by the paging module of the operating system. In the system, the processes of programs are scheduled for execution by the scheduler of the operating system. In a multiprocessor system, the scheduler manages the execution of a parallel program on multiple processors. Besides the operating system, the run-time library functions that were linked into the virtual space of a program at link time provide another vehicle for optimizing the locality of a program at run-time. For example, the run-time functions of a user-level thread library can be used for scheduling the execution of a program at run-time. Compared with the high-level compilation system and the low-level operating system, a run-time system has several characteristics:

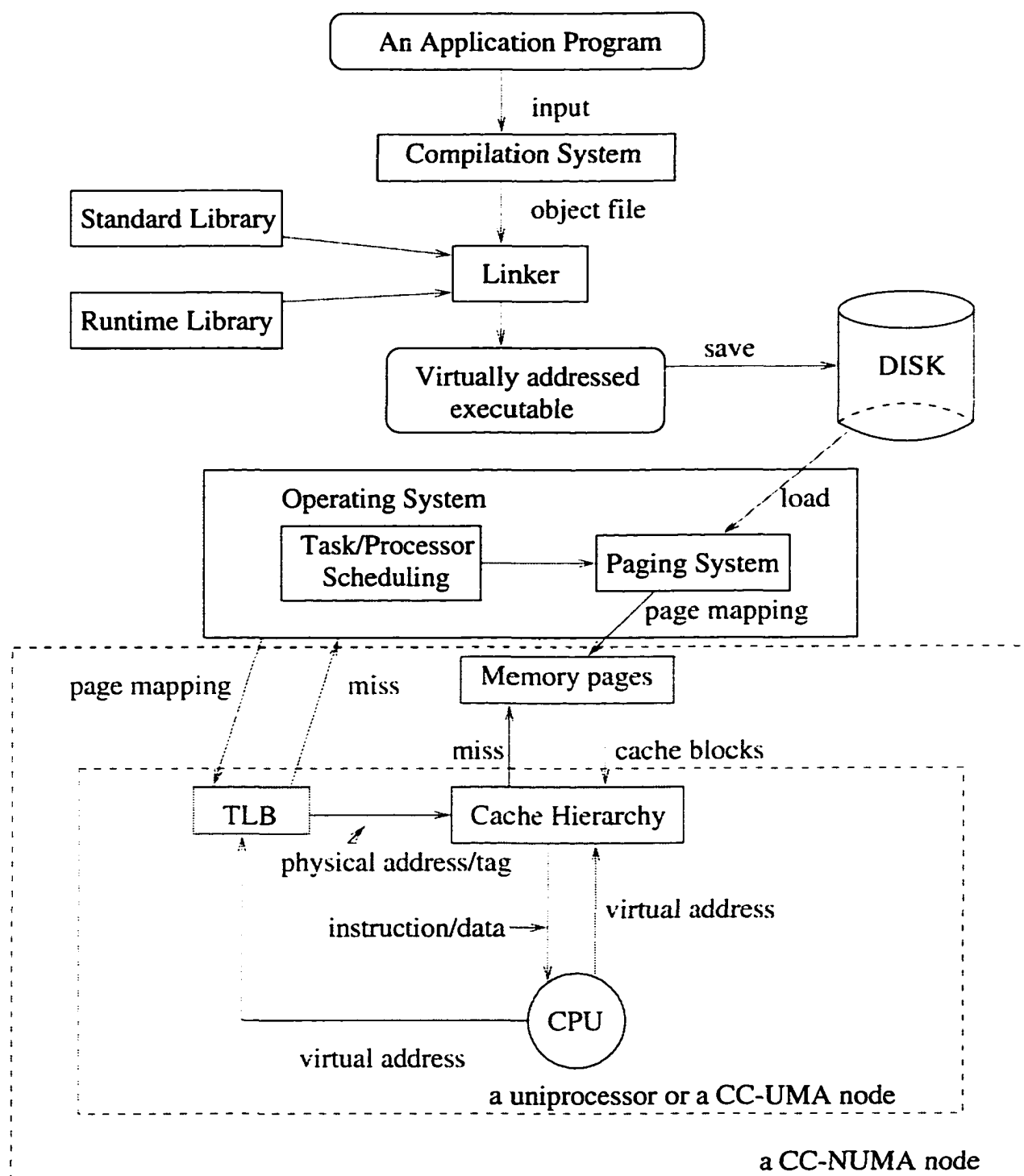


Figure 2.4: Program execution flowchart.

- It can carry high-level information into the run-time phase.
- It can capture run-time information about programs.
- It resides in the virtual space of a program, so that it can effectively handle program operations by taking advantage of application-specific information. On the other hand, during execution, architecture-specific and system-wide information can also be obtained and used for program performance optimization.

These characteristics make run-time systems very effective on improving the performance of applications, if run-time systems can take advantage of these unique features. However, a run-time system must carefully confine its overhead. In addition, an operating system is responsible for the execution of all the programs running in the system, but a run-time system usually focuses on one program.

Due to these differences between the run-time system and the operating system, we break up the system execution phase into two separate phases: the run-time system phase and the operating system phase.

5. *Hardware execution phase:* In this phase, a CPU interactively gets instructions and related data from the cache to execute. In modern computer systems, caches are addressed by two different methods: using the virtual addresses of instructions and data in a program, and using the physical addresses of instructions and data in memory. In a system with a virtual memory, the processor issues virtual memory addresses, which are dynamically translated into physical addresses. For a physically addressed cache, the processor accesses the cache with the translated physical addresses. The mapping of instructions and data onto the cache is finally determined by the paging system of the operating system. Although special-purpose hardware supports the virtual-to-physical address translation, such as TLB, it in-

creases cache-access time. An alternative to remove this bottleneck is to access the cache directly with virtual addresses. This type of cache is called a *virtually addressed cache*. In a virtually addressed cache, the mapping of instructions and data onto the cache is based on their virtual addresses, which can only be changed in the virtual space of a program. In Figure 2.4, the link from CPU with the virtual address to TLB represents an implementation of a physically addressed cache, while the link from CPU with the virtual address to the cache represents an implementation of a virtually addressed cache. In a multiprocessor system, shared data usually is mapped into multiple coherent caches. This requires that the data layout be handled very carefully.

In memory hierarchies, usually an on-chip cache is virtually addressed and an off-chip cache is physically addressed. In a cache coherent shared-memory system, coherence is maintained among off-chip caches by a hardware-implemented protocol. Excepting the addressing method of a cache, the associativity and cache block replacement method of a cache are the other two important factors that must be considered in cache locality exploitation. The caches used in modern computer systems are usually n -way set associative caches with a Least-Recently-Used (LRU) replacement policy. Direct-mapped (or 1-way set associative) caches dominate the market due to their simplicity and low design cost.

The above analyses indicate that those phases affecting the execution of an application are the places for locality to be exploited. From the implementation point of view, we can classify the locality exploitation approaches into six categories: user-level approaches, compiler-based approaches, linker-time approaches, run-time system approaches, operating system-based approaches, and hardware approaches. In next section, we overview existing work based on this classification and evaluate them from the locality types they exploit and the miss types they eliminate.

2.5 Related Work

2.5.1 User-level Approaches

Optimizing the locality of programs at the user level mainly relies on programmers' understanding of the effects of program structures and data structures on the memory performance of programs. This optimization focuses on the virtual space of a specific program, trying to minimize compulsory misses and intra-interference misses of a program. Three kinds of techniques have been developed.

A. Program Specific Approaches

With respect to a specific program, locality exploitation is based on an insightful analysis of the memory-access pattern of a program. A representative work using this approach was conducted by Lam, Rothberg, and Wolf [41]. Their work is based on a prediction model of capacity misses and compulsory misses. They study the performance of different blocked numerical algorithms, trying to find optimal blocking factors for a set of numerical algorithms. This research focused on a square blocking method. Recently, similar experiments were reconducted by Coleman and McKinley [17], based on an integrated consideration of data layout and cache organization of a matrix multiplication algorithm. Four types of misses were considered: compulsory misses, capacity misses, conflict misses among the elements of an array (called *self-conflict misses*), and conflict misses among elements of different arrays (called *cross-conflict misses*). In addition, copying, which significantly degraded performance, was shown to be unnecessary. Furthermore, Fricker, Temam, and Jalby studied the influence of cross-interferences among different array elements on blocked loops in [23]. With respect to a matrix-vector multiplication algorithm, they presented a more precise prediction model on compulsory misses, capacity misses, and conflict misses which is used to derive the optimal block size.

All the above cited references focus on the optimization of the data cache for regular numerical programs on uniprocessors, especially for nested loop structures. For irregular numerical programs, parallel programs and the other types of programs, what are the formulation of general rules for users to improve the locality of their algorithms is still form an open topic for further study.

B. Language Based Approaches

Program specific approaches indeed improve the memory performance for those specific programs. But they are program dependent. Language approaches attempt to provide explicit language mechanisms for programmers to express affinity relations, and to conduct memory layout, or task allocation on processors. High Performance Fortran (HPF) is such a language [48]. HPF provides a set of annotations for programmers to lay out data in different ways, such as interleaved, round-robin, blocked, cyclic, alignment, and dynamic redistribution. The compiler implements the specific data layout, which mainly aims at optimizing memory locality. *COOL*, designed by Chandra, Gupta and Hennessy [16], is a parallel C++ language with an emphasis on the optimization of memory performance of programs. It provides language mechanisms for users to specify affinity relations of task-to-task, task-to-data and task-to-processor, which are used to guide the scheduling of tasks at run-time. It aims at exploiting both memory locality and cache locality in a CC-NUMA system. These approaches require programmers to explicitly express affinity relations, which is more suitable for the programmers who are knowledgeable about the underlying architecture and the memory hierarchy. Existing research work focuses on data caches.

C. Profiling Approaches

For general users, it is useful to provide a tool to help them to identify program structures and data structures that cause memory performance bottlenecks. Cache profiling tools and memory profiling tools are able to provide such a service. MTOOL, designed by Goldberg and Hennessy [26], is a high-level tool to identify procedures or basic blocks that incur large memory overheads. MemSpy, designed by Gupta, Martonosi, and Anderson [27], uses procedure-level annotations to get two types of misses: compulsory misses and intra-interference misses. It also provides insight into the cause of intra-interference misses by identifying the data structures competing for space in the cache. A more detailed cache profiling tool is the CPROF program designed by Lebeck and Wood [43], which provides fine-grain source identification and data structure support. It classifies misses into compulsory, capacity, and conflict types. It uses a flexible X-windows interface to present the cache profile in such a way that helps the programmer determine cache performance bottlenecks. Recently, a tool, named CVT was developed by Deijl, Teman, Granston, and Kanbier [21] to visualize the cache content and to show its evolution during the execution of a piece of code. These tools only help programmers understand the memory performance of a program. The decision of how to use the information to exploit cache locality must be made entirely the users. So far, these tools are designed for sequential programs.

2.5.2 Compiler Approaches

To relieve programmers of the heavy burden of conducting locality analysis at the user-level, compiler approaches provide important solutions. The main idea behind a compiler's exploitation of locality is to transform the data layout and computation order of a program based on compilation information. On a uniprocessor, locality optimizations focus on eliminating compulsory cache misses and intra-interference cache misses. On a

multiprocessor, locality exploitation techniques must minimize cache invalidation misses and remote accesses, as well as those misses occurring in a uniprocessor system. Compilation approaches are static in the sense that they optimize the locality of applications based on statically known information at compile-time. The applications with dynamic memory-access patterns challenge compiler techniques. Regarding the locality optimization at compile-time, there are two large volumes of work: one focuses on uniprocessor systems, and the other focuses on multiprocessor systems.

A. Locality Optimizations For Uniprocessor Systems

The common transformations used for arranging data layout are: *array merging*, where arrays accessed in a loop are merged together to increase spatial locality, *structure and array packing*, where structures and arrays accessed in a loop are packed together to increase spatial locality, *padding*, which changes the relative distances among data structures or elements of a structures to eliminate conflict misses, and *structure aligning*, which aligns structures with respect to a block mapping so that a structure is spanned over a minimal number of cache blocks. Common transformations used for reorganizing computations are *loop permutation* (or *loop interchange*), which exchanges inner loops with outer loops to change data-access patterns, *loop fusion*, which combines multiple loops into one to increase spatial locality, *loop distribution*, which divides one loop into multiple loops to eliminate conflict misses, *loop reversal*, which legally reverses the order in which the iterations of a nested loop execute, and *tiling* (or *blocking*), which blocks a large iteration space of a nest loop into small parts so that each can fit in the cache. Detailed explanation on these transformations can be found in [81]. Compiler based locality exploitation techniques try to find a sequence of transformations for each program structure so that cache misses are minimized.

In relatively early times, references [22] and [25] focused on evaluating data

locality for a given loop permutation. In order to find the loop permutation which yields the best data locality, the proposed techniques may consider up to $n!$ loop permutations. Wolf and Lam [82] classified data reuse into four categories: *self-temporal reuse*, *self-spatial reuse*, *group-temporal reuse*, and *group-spatial reuse*. Then a prediction model to quantify both reuse and locality was developed to guide the unimodular transformation of a loop by interchange, skewing, reversal, and tiling. Along this track, more recent work was done by McKinley, Carr, and Tseng [54]. They derived a simple cost model to compute temporal and spatial reuse of cache lines. The derived model is simpler than Wolf and Lam's model given in [82], without precise modeling of group references and without consideration of the order of outer loops in a nest loop. Based on the cost model, they derived the application of compound transformations, which consist of loop permutation, loop fusion, loop distribution, and loop reversal. One major contribution of this work is that it demonstrated the usefulness of the proposed technique for a large collection of scientific programs and kernels. Finally, they questioned if a more precise cache model could yield performance improvements in practice for real applications. These optimizations target on a data cache, and they put more emphasis on program structure transformation than on data layout. It is still an open question how to lay out data systematically for improving the cache locality of applications while transferring program structures.

B. Locality Optimizations For Multiprocessor Systems

Compared with the locality optimization on a uniprocessor, the locality optimization on a multiprocessor must trade off parallelism and locality while eliminating cache misses. The early work in [25] took parallelism into consideration while trying to optimize locality of applications. It attempted to insert parallelism at the outermost possible position. But it neither considered how the parallelism affects the locality nor if an interchange would

improve the granularity of parallelism. Kennedy and McKinley [39] proposed an approach which attempted to combine the benefits of locality optimization and parallelism optimization. Guided by a simple locality prediction model, their approach produces data locality at the innermost loop and places parallelism at the outmost loop. However, their model is a uniprocessor cache model, which does not take into consideration the effects of false sharing and true sharing. So, invalidation misses, an important kind of cache misses in a multiprocessor system, were not considered. Manjikian and Abdelrahman [50] proposed some techniques which were used in loop fusion to optimize the locality of loops. However, they did not address how their techniques could be used with other loop transformations. In order for a compiler to optimize locality and parallelism systematically, Li and Pingali [46] proposed a linear transformation technique, called *access normalization*, which restructures loop nests to exploit both locality and block transfers. This technique is based on the framework of invertible matrices and integer lattice theory, which is a generalization of Banerjee's framework of unimodular matrices [8]. One limitation of this approach is that it only focuses on perfect nests or nests that can be made perfect with conditionals.

Because the communication overhead caused by false sharing and true sharing is a major performance bottleneck for multiprocessor applications, some techniques have been developed to handle this overhead. Agarwal, Kranz and Natarajan [3] addressed this issue by proposing a theoretical framework for automatically partitioning parallel loops to minimize cache coherency traffic. However, their approach does not exploit locality in each partition. It only reduces invalidation misses and remote memory accesses.

The above techniques focus on using program optimizations to improve the locality of applications. Data layout is another important factor affecting the locality of applications. Jeremiassen and Eggers [34] addressed this issue using a compiler approach. They used three separate compiler analysis stages to pinpoint susceptible data structures that would cause false sharing. Then, some data transformations were applied at appro-

priate points. However, all the data transformations were done with respect to a given program partition. Ideally, data transformations and program transformations should be conducted together. Targeting this problem, Torrellas, Lam, and Hennessy [73] proposed four data layout techniques to reduce false sharing. However, the application of their data transformations was studied independently of the program transformation.

An integration of data decomposition and computation decomposition was conducted by Anderson and Lam [6]. They modeled decomposition as affine functions and structured decompositions into three components: *partition*, which determines which array elements and iterations are local to a single processor, *orientation*, which gives the correspondence between the data and computation dimensions and the processor dimensions, and *displacement*, which specifies the offsets of the array elements and iterations with respect to the processors. This approach tries to find a static decomposition that exploits the maximum degree of parallelism available in a program such that communication is minimized. It is aimed at achieving maximal parallelism and minimal communication overhead (or memory locality in a CC-NUMA). However, their approach neither addressed the load balance issue nor exploited the cache locality. In addition, data transformations are actually very helpful for data decomposition as shown in [34, 73]. More recently, Anderson, Amarasinghe, and Lam [5] combined the optimizations of parallelism, communication, and data layout. Their approach consists of two steps: decomposition of computation and data, and restructure of data layout. However, this work does not address the load balance issue in the decomposition of computation and data. In addition, the applicability of data transformations should be evaluated for a wider range of applications with different memory-access patterns. Using a compiler approach to exploit locality of applications is particularly attractive, because a compiler can conduct insightful analyses. Besides the issues pointed out above, which need further investigation, compiler approaches only focus on the optimization of a program in the virtual space. For a physically addressed cache, underlying operating system interferes with

the cache-access patterns of applications. Recently, reference [12] proposed a compiler-directed page coloring method to improve cache locality on a physically addressed cache. Because the implementation of this approach finally relies on the underlying operating system, we will evaluate it in the category of system approaches. In addition, all compiler approaches cited above only focus on the data cache.

2.5.3 Link-time Approaches

Besides data locality optimization, instruction locality optimization is equally important for improving the memory performance of applications. The main idea of exploiting instruction locality is to reorganize program code. A static linker is a good place to do code reordering because it has a global picture of what static codes are going to be linked. To reorder program codes, a linker must get sufficient information on programs from the compiler phase. However, the dynamically linked library functions and system calls are out of the control of a static linker. Current work focuses on the optimizations in a static linker.

Hwu and Chang [33] improved instruction cache performance using inlining, basic block reordering, and procedure reordering. Based on a call graph with weights produced by profiling, their algorithm maps procedures to the address space by traversing the call graph along heavily weighted edges in depth-first order. The depth-first traversal may lead to an unimportant path in the control graph, because the traversal is guided by local knowledge, the weights of currently available edges, and not the global knowledge in the call graph. Similarly, Pettis and Hansen [59] presented a number of techniques to improve code layout: basic block reordering, procedure splitting, and procedure reordering. Their algorithm uses a closest-is-best strategy to perform procedure reordering, which starts with the heaviest executed call edge in a program call graph. The above two approaches improve the locality of an instruction cache. A disadvantage is that both

methods do not consider the architectural information of an instruction cache, which is necessary for a more precise reordering technique.

Some techniques do conduct code reordering by considering the underlying cache architecture. McFarling [52] improved instruction-cache performance by not caching infrequently used instructions and by reordering codes. Based on a control flow graph with basic block, procedure, and loop nodes, the author attempted to partition the control graph by focusing on the loop nodes, so that each partition tree has a code size smaller than the instruction cache. The limitation of this approach is that a control graph may not always be partitioned in this way. For operating system intensive workloads, Torrellas, Xia and Daigle [75] proposed an algorithm for code layout by taking into consideration both the cache size and the code popularity. The algorithm partitions the operating system code into executed and non-executed parts at the basic block level, and then creates sequences of basic blocks from the executed code controlled by a decreasing threshold value. For a given threshold value, all the basic blocks with larger weights are removed and put into a sequence. All the blocks in a sequence are laid out together in the address space. The procedure is repeated until all frequently executed basic blocks have been put into sequences. The algorithm maps the most frequently executed sequence into a special area in the cache and the rest of the sequences to cache areas that avoid the special area. The non-executed basic blocks are used to fill the final gaps remaining in the cache. Because this approach targets operating system workloads, it does not consider the relations between blocks, which may prevent this approach from being applied by user programs. Recently, Hashemi, Kaeli, and Calder [29] proposed an algorithm to lay out procedures based on more detailed information about the cache. They classified procedures as popular procedures and non-popular procedures. First the cache blocks of a cache are marked into different colors, and a call graph with call numbers as the weights of edges is constructed for popular procedures. Then, their algorithm tries to eliminate conflict misses by coloring two adjacent nodes of the call graph in a step-by-

step method. At each step, the two nodes connected by the heaviest weight edge are colored to avoid conflicts in the cache. The two nodes are then merged into a node in the call graph. This procedure is repeated until all procedures are colored. In the end, the non-popular procedures are used to fill the gaps left by the procedure coloring. This approach gets better performance than previous work due to its deep exploitation of cache information. However, this approach may be further improved, because it only takes into consideration procedure layout. Sometimes block layout is necessary. In addition, the optimal code layout problem is an NP-Complete problem which leaves some room for improvement. Another interesting issue is to investigate how far existing techniques are from the optimal solution.

2.5.4 Run-time Approaches

Although a compiler can do complicated transformations, it performs poorly for programs with dynamic structures, and it cannot predict the effect of the underlying system. To remedy these drawbacks of compiler, a run-time library is usually a complementary solution. The major advantage of the run-time approach for exploiting locality is that it can take advantage of both static information from an application or a compiler and dynamic information from the underlying system to predict the cache-access pattern of an application. However, the run-time approach is strictly constrained by the low-overhead requirement. So, algorithms in a run-time system must be relatively simple and effective.

In order to achieve balanced partitioning on parallel loops, run-time loop scheduling algorithms have been intensively studied previously. In order to minimize remote accesses in a CC-NUMA system, Markatos and Leblanc [51] proposed an affinity loop scheduling algorithm. Their approach allocates a local task queue to each processor and keeps processors busy with the local tasks as much as possible. This scheduling algorithm only exploits processor affinity to increase data reuse when parallel loops are repeatedly

executed without changes in the memory-access patterns. However, data affinity is very important to improve memory locality. Li, Tandri, Stumm, and Sevcik [45] proposed a loop scheduling algorithm with consideration of data affinity. Their approach tries to allocate loop iterations local to their data so that remote memory accesses can be minimized. The run-time system must get information on data layout from the high-level compiler. Their approach does not exploit processor affinity, namely the cache locality.

The integrated exploitation at run-time of cache locality and memory locality was studied by Chandra, Gupta and Hennessy [16] in the context of the design of *COOL*. Based on user-specified affinity relations, their run-time system schedules the executions of tasks to exploit processor affinity and data affinity. A limitation of this approach is that the quality of locality optimization totally depends on the programmer. Recently, Philbin, Edler, Anshus, Douglas, and Li [60] proposed a parallel threading approach to exploit cache locality of sequential programs on a uniprocessor. With respect to loops that can be parallelized, fine threads are created for loop iterations by carrying some hints about their access data. Their run-time system uses a cache-size based square blocking method to cluster threads so that the threads accessing the same data execute consecutively to reuse cache data. This approach does not require programmers to specify affinity relations explicitly like the approach presented in [16].

This area has been paid less attention so far. More intelligent run-time exploitation techniques are expected to be designed for those programs with dynamic or irregular memory-access patterns.

2.5.5 Operating System Based Approaches

An operating system is mainly responsible for the management of computations and computation resources. The memory management, task scheduling, and processor scheduling (on a multiprocessor) of an operating system are the components that directly affect the

locality of applications. The memory paging system maps virtual pages of a program into physical pages, which determines the mapping of instructions and data to the cache on a physically addressed cache. In a CC-NUMA system, page mapping must consider how to place virtual pages in order to maximize memory locality as well as cache locality. This is correlated with the scheduling of tasks and processors. The task scheduling algorithm and the processor scheduling algorithm determine where and when a task executes, which affects the memory locality and cache locality of applications. So, operating system based locality optimizations are exploited respectively in the memory management system and the scheduling algorithms of tasks and processors. Compared with the locality exploitation at high levels, the locality exploitation in the operating system kernel has a major advantage of being able to perform system-wide locality optimizations for all the applications in the system. This feature is important for the performance of a time-sharing system.

A. Page Management

Traditionally, when a program is loaded into memory, the operating system maps the virtual pages of the program into memory pages by randomly selecting from a pool of available pages. This random page mapping tends to cause two frequently accessed pages to be mapped onto the same cache area. This generates either intra-interference cache misses, if the two pages belong to one program or inter-interference cache misses, if the two pages belong to two different programs. Additionally, on a CC-NUMA system, an inappropriate page management could cause a large number of remote memory accesses. So, care should be taken for page management. On the optimization of cache locality, two classes of page mapping algorithms have been proposed: *static algorithms* that work at page-in time and never change the mapping of a virtual page to a physical page, and *dynamic algorithms* that dynamically update the mappings between virtual and physical

pages in order to minimize cache conflicts.

Regarding static page mapping, Kessler and Hill [40] proposed four policies: *page coloring*, *bin hopping*, *best bin*, and *hierarchical*. To manage the mapping of memory pages into the cache, the cache is divided by page size and cache pages are distinguished with different colors. Similarly, each memory page is colored by the color of the cache page into which it will be mapped. All memory pages with the same color are organized into a page bin. When virtual pages are paged into memory, different policies are used to exploit different types of locality. The page coloring policy maps consecutive virtual pages to consecutive colors so that pages close together in the virtual space do not conflict in the cache. The bin hopping policy cycles through the available colors sequentially as pages are faulted in, so that pages mapped close in time tend to be placed in different bins, regardless of whether these pages belong to the same virtual space or to different virtual spaces. The advantage of the bin hopping policy over the page coloring policy is attributable to the fact that the former uses a global count to remember the most recently used color. The best bin policy uses more global knowledge in its decision making. This algorithm has linear complexity in the number of bins. The hierarchical policy tries to reduce the complexity of the best bin policy by using a binary tree so that the search of the best bin can finish in log complexity of the number of bins. The first two policies have constant complexity. They are adopted by some modern operating systems (for example, the IRIX 5.3 uses page coloring and Digital UNIX uses bin hopping). Recently, Bugnion, Anderson, Mowry, Rosenblum and Lam [12] proposed a new approach to further reduce intra-interference misses on a multiprocessor. Their approach gets the compiler to create a summary of the array-access patterns of a program, the run-time system to color the virtual pages of the program based on machine-specific parameters (i.e., the number of processors, the cache configuration, and the page size), and the operating system to honor the run-time page coloring as much as possible. Because this approach carries application information down to the operating system, it can make a more precise

page mapping than the page coloring and the bin hopping. They demonstrated the advantage of their approach only on a dedicated multiprocessor. Whether this approach outperforms page coloring and bin hopping on time sharing systems still needs further investigation. Trying to honor the best mapping of an application does not guarantee a performance improvement for a time sharing system.

Compared with static page mapping policies, a dynamic page mapping policy can adjust page mapping based on run-time information. A hardware-assisted dynamic page mapping policy was proposed by Bershad, Lee, Romer, and Chen [9]. They proposed an inexpensive hardware device, called the Cache Miss Lookaside (CML) buffer, to detect conflicts by recording and summarizing the history of cache misses, and a software policy, called the *sequential-target* policy, within the operating system's virtual memory system to remove conflicts by dynamically remapping pages whenever large numbers of conflict misses are detected. This approach enables a direct-mapped cache to perform nearly as well as a two-way set associative cache of equal size and speed. Meanwhile, they further investigated the possibility of using standard hardware to recolor pages in [62]. Their approach is aimed at using the TLB and the cache miss counter to locate possible cache conflicts. They showed that a dynamic page mapping policy using standard hardware can improve upon the performance of a static policy, but is not as effective as special-purpose hardware, such as an associative cache or an CML buffer.

All the above research work except [12] focuses on uniprocessor systems. On a multiprocessor system, the recoloring of shared pages should consider its effect on multiple caches. How these approaches are extended to a time sharing multiprocessor system is a current research topic.

On a CC-NUMA system, the page management system must take care of memory locality as well as cache locality. Early research on this problem was done by Chandra, Devine, Verghese, Gupta, and Rosenblum in [15]. The other early work on this topic was in the context of non-cache-coherent NUMA machines, such as [11] and [42].

Their approach migrates data pages based on TLB misses. Recently, Verghese, Devine, Gupta, and Rosenblum [80] proposed a more effective algorithm to reduce remote memory accesses based on page migration and page replication. In their approach, pages are classified into three groups based on cache misses or TLB misses, the first group consisting of pages primarily accessed by a single process, the second group consisting of pages most read by multiple processes, and the third group consisting of pages both read and written by multiple processes. Page migration is used for the first group of pages. Page replication is used for the second group of pages. The effectiveness of the proposed approaches were demonstrated for a wide range of applications in the context of the space-sharing scheduling. Cache miss counting was shown to be a more precise measure than TLB miss counting to differentiate pages.

B. Scheduling

When a processor holds a part of the working set of a task in its caches (*cache affinity*) or local memory (*memory affinity*), the processor is said to be in affinity to the task. Exploiting processor affinity tends to improve the execution performance of tasks by causing less cache misses and/or remote memory accesses. The affinity of a task to a processor is generated by three sources: (1) a previous execution of the task on the processor that causes the working set to be cached (resulting in cache affinity), (2) page allocation and migration, which get the working set of the task to reside in the local memory of the processor (resulting in memory affinity), (3) executions of affinity tasks that have similar working sets (resulting in cache affinity). The first affinity source can be exploited alone by processor and task scheduling algorithms. The exploitation of the second affinity source needs to take into consideration the effects of page management policies. The third affinity source usually is hard for a pure OS-based approach to exploit due to the lack of knowledge on tasks' working sets. The challenge of exploiting locality in

the system schedulers comes from the requirement of a set of conflict goals. For example, context switching is aimed at achieving fairness among concurrent tasks, which tends to weaken cache affinity. This has been shown to have significant effect on cache locality [56]. So, schedulers must carefully trade off performance and other goals.

Based on a formal model and measurements, Squillante and Lazowska [65] showed that even exploiting the simplest forms of processor-cache affinity could potentially provide significant improvements over ignoring this affinity on a time-sharing multiprocessor. They proposed six scheduling policies to investigate the benefit of taking advantage of locality. Regarding the implementation of the algorithms, two approaches were suggested: a *queue-based* approach that considers the organization and use of task queues to incorporate processor-cache affinity in scheduling decisions, and a *priority-based* approach that considers augmenting the system's priority discipline with processor-cache affinity information. Including this information in the priority calculation allows the scheduler to balance a task's affinity with other scheduling criteria. Torrellas, Tucker, and Gupta [74] further investigated the benefits of affinity for a wider mix of workloads in more common time-sharing systems. They used a priority-based technique to implement a simple affinity scheduler. They showed that affinity scheduling could achieve a significant reduction in execution time and cache misses. However, these scheduling policies are more suitable for a CC-UMA than a CC-NUMA because they do not consider page placement policies.

Because context switching is a factor degrading the cache performance of applications, Black [10] proposed a space partition technique to reduce the interference among parallel computations on a multiprocessor system. His approach partitions the machine into sets of processors, each of which executes a single parallel application. Other references [15, 28, 76] showed that the best performance was obtained by partitioning the available processors among concurrently executing jobs rather than by rotating the processors among them in a time-slicing manner (time-sharing). Vaswani and Zahorjan [78]

investigated whether cache affinity exploitation could improve system performance in the context of space partition. Their evaluation concluded that affinity scheduling provided little benefit under current conditions but will have a modest effect on the much faster machines of the future. This research had emphasis on a CC-UMA machine. Based on current development trend of architectures, Symmetric Multi-Processors (SMPs) with 2-4 processors will be the only CC-UMA systems [19], which are time sharing systems. Space-partition is more likely to be used in a CC-NUMA system, where affinity exploitation must be considered in an integrated manner in both the scheduling policy and the page management policy. Regarding this, Chandra, Devine, Verghese, Gupta, and Rosenblum [15] showed a significant performance achievement by locality exploitation in the operating system kernel.

Although current affinity scheduling algorithms are effective to improve system performance, they only exploit simple affinity hints, such as the place a task last ran and the last task a processor executed. To further improve the system performance using affinity scheduling, more precise affinity hints should be exploited.

2.5.6 Hardware Approaches

Although many novel cache designs (see e.g., [4], [86] and [30]) have been proposed recently, the caches used in modern computers are direct-mapped caches and set-associative caches. Regarding cache coherent multiprocessor systems, the design of cache coherent protocols is critical for reducing invalidation cache misses. Recently, the research work on this topic has been surveyed in [70] and [72]. Improving cache performance has been and remains a very active research area. Hennessy and Patterson [30] have given an overview on existing work. Here, we only focus on those hardware techniques that are aimed at reducing the miss rate of a single direct-mapped cache or set-associative cache. This restricts our overview to a very narrow area.

In order to reduce the sensitivity of direct-mapped caches to conflicts, Jouppi [37] proposed a small full-associative cache, called the *victim cache*, between a cache and its refill path. A victim cache contains only blocks that are evicted by misses and are checked at each miss. This work showed that a four-entry victim cache reduced 20% to 95% of the conflict misses in a 4-KB direct-mapped data cache. Another novel ideal to reduce the cache miss rate is *cache bypassing*, which determines cache block replacement based on dynamic information. McFarling [53] used a small finite state machine to recognize the common instruction reference patterns so that those instructions whose storing will harm performance are passed through the cache without being stored. This technique reduces 33% of the miss rate for a 32 KB direct-mapped instruction cache. On a direct-mapped cache, this approach still requires augmenting each set with two additional bits. This technique targets instruction caches.

Recently, Johnson and Hwu [36] proposed another adaptive bypassing technique for data caches. The physical space of a program is divided into microblocks. A hardware Memory Address Table (MAT) was introduced between caches and memory, which records the accesses on a specific microblock. The main idea is to combine LRU and MAT-guided bypass. The difference between the bypass buffer and a victim cache is that the bypass buffer only holds the bypassing data, not the data block containing the bypassing data. The victim cache holds the cache blocks evicted from the cache. Cycle-by-cycle simulations showed the MAT scheme outperformed large victim caches, even for a finite-size MAT of a similar hardware cost and less associativity. The size of Microblock is mainly determined by a detailed analysis of the memory references of an application. More applications should be further examined.

2.5.7 Summary

We have overviewed related work on locality exploitation at six different system levels, and addressed their merits and limits. The major advantage of the user-level approaches is that users can improve the locality of a program by changing the algorithmic structure of a program. However, current work is restricted to simple program structures. Complicated program structures, such as programs with irregular computing patterns, dynamic data-accesses, or dynamic data dependence, are very difficult for users to tune by hand. These approaches place an unacceptable burden on users. Our run-time technique provides a simple interface that can be directly used by users. Knowledge of the details of the complicated run-time optimizations is not required.

At the compile-level, the overview has shown that a large amount of interesting work have been done. The main idea of the current work is to find a sequence of proper transformations that will result in an improved memory performance. The major challenge in compiler-based techniques is dealing with the irregular computational patterns, dynamic memory-access patterns and dynamic data-dependence patterns. Dealing with this challenge is the goal of our run-time technique.

The linker-level approaches are aimed at optimizing the instruction locality of a program. Our run-time technique targets optimizing the data locality of a program. The data locality is more difficult to optimize than the instruction locality because instruction execution patterns are much more regular than data-access patterns. At the run-time system level, most existing work only exploits processor *affinity* in scheduling and partitioning schemes. Some research efforts improve the memory performance of an application in uniprocessor systems or in multiprocessor systems using user-provided affinity information that is not available in complex applications. Our run-time technique exploits the data locality on a symmetric multiprocessor, based on the affinity information automatically exploited at run-time.

There is also a large volume of works at the system level and at the hardware level. This work is aimed at improving system-wide memory performance, not the memory performance of a specific program, and are complementary to the program-specific optimizations.

Chapter 3

Cache-Locality Optimization Models

The dynamic nature of application programs and computer systems makes the cache-locality optimization problem complicated. In this chapter, we present the programming model, the targeted multiprocessor architecture and a framework of solutions for the cache-locality optimization.

3.1 Programming Model

In applications, frequent data accesses usually occur in loop structures [55]. Thus, the loop is a major programming structure that would have a significant impact on the memory performance of an application. We target the following class of loop structures, which are commonly used in scientific applications.

The program structures addressed in this dissertation are nested loops as shown in Figure 3.1. All the programs presented in this dissertation are in C-language format consistent with the C-language implementation of our run-time system. In Figure 3.1, l_j and u_j are, respectively, the lower bound and upper bound of loop index variable i_j for $j = 1, 2, \dots, k (k \geq 1)$. These two bound variables are often functions of the outer loop

```

for ( $i_1 = l_1; i_1 < u_1; i_1++$ )
  for ( $i_2 = l_2; i_2 < u_2; i_2++$ )
     $\vdots$ 
    for ( $i_k = l_k; i_k < u_k; i_k++$ )
      B;

```

Figure 3.1: Data-independent nested loop.

index variables, i_1, i_2, \dots, i_{j-1} , and are determined at run-time. The loop body **B** is a set of statements where the statements can also be loops. An execution instance of loop body **B** can be considered as a fine-grained task to be expressed as $\mathbf{B}(v_1, v_2, \dots, v_k)$, where v_j is the value of index variable i_j for $j = 1, 2, \dots, k$. The proposed run-time system in this dissertation targets the tasks whose memory-access patterns are determined by run-time data. The memory-access patterns of this type of applications are difficult to exploit at compile-time.

The condition that the above nested loop must satisfy is defined as follows. All the execution instances of the loop body **B** are data-independent, i.e., for any two instances, denoted as instance $\mathbf{B}(v_1^\Delta, v_2^\Delta, \dots, v_k^\Delta)$ and instance $\mathbf{B}(v_1^\nabla, v_2^\nabla, \dots, v_k^\nabla)$, the following condition is valid:

$$\begin{aligned}
out(\mathbf{B}(v_1^\Delta, v_2^\Delta, \dots, v_k^\Delta)) \cap out(\mathbf{B}(v_1^\nabla, v_2^\nabla, \dots, v_k^\nabla)) &= \emptyset \wedge \\
out(\mathbf{B}(v_1^\Delta, v_2^\Delta, \dots, v_k^\Delta)) \cap in(\mathbf{B}(v_1^\nabla, v_2^\nabla, \dots, v_k^\nabla)) &= \emptyset \wedge \\
in(\mathbf{B}(v_1^\Delta, v_2^\Delta, \dots, v_k^\Delta)) \cap out(\mathbf{B}(v_1^\nabla, v_2^\nabla, \dots, v_k^\nabla)) &= \emptyset
\end{aligned} \tag{3.1}$$

where notations *out* and *in* represent respectively the output variable set and input variable set of an instance [81].

Although a more general class of loop structures carries some data-dependence, the proposed techniques in this dissertation can also be applied to the class by combining

data-dependence analysis techniques. This dissertation proposes a new cache-locality optimization technique by focusing on data-independent loop structures. In the conclusion, we will discuss how to apply our technique to program structures with data dependence.

3.2 SMP Architecture Model

The targeted shared-memory system in this study has a symmetric memory system as shown in Figure 1.1. This symmetric multiprocessor shared-memory system model, called SMP, has been widely used in existing commercial multiprocessor systems, such as the HP/Convex Exemplar S-class [7], the Sun SPARCcenter 2000 [14] and the Hyper-SPARCstation-20, the SGI Challenge [24], and the DEC AlphaServer [66].

In an SMP, each processor has the same access latency to the shared memory and has a hierarchical cache system in each processor, which usually consists of two-level caches: an on-chip first-level cache and an off-chip secondary cache. A hardware cache coherence protocol is implemented among the secondary caches to guarantee the consistency of accesses to shared data. The interconnection network may be a bus, a mesh, or a crossbar architecture. A snooping cache coherence protocol is typically used in a bus-based shared-memory system [14, 24, 66], and a home directory based cache coherence protocol is used for a mesh- or crossbar-based shared-memory system [7].

The caches implemented in most existing SMPs are k -way set associative caches (here a 1-way set associative cache refers to a direct-mapped cache). The associative degree, k , tends to be less than 4. The cache block replacement policy of a cache is LRU. The principle of addressing a k -way set associative cache is illustrated in Figure 3.2. The block address of address a is given by a/b where b is the block size (which is 2^d in the example). Then the lower r bits of the block address gives the set number. When a block is being brought into a cache set and there is no empty block, the least recently used block in the set is replaced by the LRU replacement policy.

Because the secondary cache usually is physically addressed, the caching pattern of shared data at this cache level cannot be determined at compile time. In this case, a run-time approach is the only way to capture the cache reference pattern of an application. In an SMP system, exploiting the locality of applications to take good use of caches has three performance implications: reducing memory-access latency, cache coherence overhead, and decreasing shared-memory access contention.

Besides SMPs, Cache-Coherent Non-Uniform Memory Access (CC-NUMA) system, such as the KSR shared memory system [38] and the Stanford DASH system [44], is a different type. The major difference between a CC-NUMA and an SMP is that a CC-NUMA has a distributed shared memory system, where each processor has a local shared memory module. In a CC-NUMA system, the access latency from a processor to its local shared memory usually is significantly smaller than that from the processor to the local shared memory of another processor, called *remote memory*. Although the cache locality exploitation method proposed in this dissertation can be used in a CC-NUMA system, it must be extended by considering the special access patterns in the distributed shared-memory. This needs more cooperation between a compiler and the run-time system to capture the data layout pattern of an application. In this dissertation, we focus on an SMP system. The extension to a CC-NUMA will be discussed in Chapter 8.

3.3 Cache Locality Exploitation Model

The goal of exploiting the cache locality of a parallel application in a shared memory system is to maximize data reuse in a single cache and to minimize data sharing among multiple caches for the purpose of minimizing cache coherence overhead. For a given data memory layout, how many data items can be reused in the cache depends on how the data is referenced. How many data accesses among multiple caches interfere with one another depends on how the application program is partitioned.

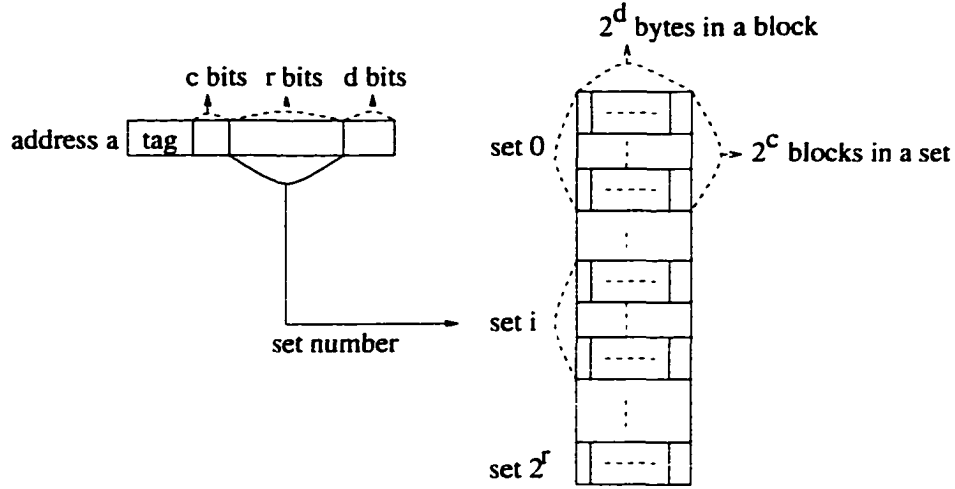


Figure 3.2: Addressing in an k -way set associative cache where the number of sets is 2^r , block size is 2^d , and $k = 2^c$.

In this section, we formally model the locality optimization problem for a given data memory layout on uniprocessors and multiprocessors. Using formal models, we try to find an optimal execution which maximizes the data reuse, and we try to give the complexity of finding such optimal solutions. This formal model provides a guideline to design and to analyze practical cache-locality optimization methods.

3.3.1 Locality Optimization of Sequential Executions

For a given memory layout, the number of memory accesses in a sequential task is constant. Because locality optimization is aimed at reducing the number of memory accesses of an application, we can quantitatively define the reusability of a sequential execution $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n$ of n sequential tasks: t_1, t_2, \dots, t_n , as follows.

Definition 1 For n ($n > 1$) sequential tasks: t_1, t_2, \dots, t_n , the reusability, denoted $\text{Reuse}(t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n)$, of sequential execution $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n$ is defined as:

$$\text{Reuse}(t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n) = \sum_{i=1}^n \text{Mem}(t_i) - \text{Mem}(t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n),$$

where $\text{Mem}(e)$ gives the number of memory accesses in sequential execution e .

Here, we present a precise method to calculate the reusability so that the locality optimization problem can be rigorously formalized.

Let t_1, t_2, \dots, t_k be k data independent tasks that satisfy the condition described in equation (3.1). Let $\text{Dset}(t_j)$ denote the set of addresses of data accessed by task t_j . In the following, an address is used to abstract the data at that address, so data caching can be abstracted as address caching. Because a majority of cache architectures in commercial computers are k -way set associative caches with a LRU replacement policy [30], we only focus on this type of cache architecture.

To capture the data reuse in a cache between two different tasks, we define the following cache-reusable relation.

Definition 2 Two addresses $a_i (\in \text{Dset}(t_i))$ and $a_j (\in \text{Dset}(t_j))$ are cache-reusable, (denoted by $a_i @ a_j$), if both addresses a_i and a_j reside in the same memory block.

It is not difficult to verify that relation $@$ is reflexive, symmetric, and transitive, so $@$ is an equivalence relation. The intuitive meaning of the cache-reusable relation is that two cache-reusable addresses reside on the same cache block so that an access to either one will bring the other one into the cache. The cache-reusable relation is an abstraction of two types of localities: temporal locality and spatial locality.

Based on relation $@$, the maximal set of addresses of task t_j , whose data blocks may be reused by a next task t_i in the execution sequence, is

$$\{a | a \in \text{Dset}(t_j) \wedge \exists b \in \text{Dset}(t_i) (a @ b)\}.$$

However, it is not guaranteed that all the memory blocks accessed by a task will remain in the cache after its execution. Because some of the memory blocks will be mapped onto the same cache line, only the last memory block brought into a cache line remains after

the execution of a task. To analyze the data reuse precisely in a cache, it is essential to differentiate clearly the cache interference among the memory access addresses within a task. We need to distinguish those special memory addresses of a task which affect data reuse in a cache.

Here, we introduce two other concepts: post-distance and pre-distance.

Definition 3 Let $a_{i1} \rightarrow a_{i2} \rightarrow \dots \rightarrow a_{in}$ be the address sequence of memory accesses of task t_i where $a_{ij} \in Dset(t_i)$ for $j = 1, 2, \dots, n$. The block address sequence is given by $a_{i1}/b \rightarrow a_{i2}/b \rightarrow \dots \rightarrow a_{in}/b$, where b is the block size.

1. The post-distance of address a , denoted as $\text{postd}(a)$, is defined as: (1) ∞ (infinitely large), if a has the same block address as another distinct address that occurs after the last occurrence of a in the address sequence; otherwise, (2) the number of distinct block addresses that occur after the last occurrence of the block address of a in the block address sequence and have the same cache mapping set as a .
2. The pre-distance, denoted as $\text{pred}(a)$, of address a is defined as: (1) ∞ (infinitely large), if a has the same block address as another distinct address that occurs before the first occurrence of a in the address sequence; otherwise, (2) the number of distinct block addresses that occur before the first occurrence of the block address of a in the block address sequence and have the same cache mapping set as a .

Figure 3.3 shows an example for the calculation of postd and pred for a short memory access sequence of a task. Figure 3.3(a) shows three memory blocks containing memory addresses from 0 to 5. Figure 3.3(b) illustrates a 2-set cache with a block size of 2. Figure 3.3(c) shows the memory access sequence, the corresponding block address sequence, and cache set mappings of addresses. The sequence execution order is from left to right. The calculation results of postd and pred are given in Figure 3.3(d).

Based on postd and pred , we distinguish two classes of memory addresses within a task: frontier addresses and back addresses.

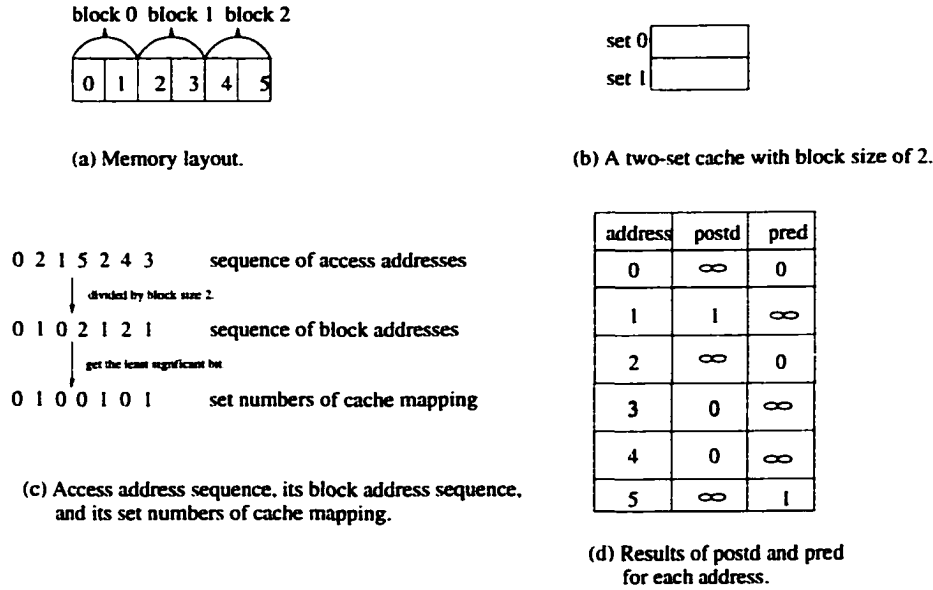


Figure 3.3: An example for the calculation of *postd* and *pred* for each memory access address in an address sequence.

Definition 4 For any sequential task t on a k -way set associative cache, its frontier address set, denoted as $\text{frontier}(t)$, and back address set, denoted as $\text{back}(t)$, are defined as follows:

$$\text{frontier}(t) = \{a | (a \in \text{Dset}(t)) \wedge (\text{pred}(a) < k)\} \quad (3.2)$$

$$\text{back}(t) = \{a | (a \in \text{Dset}(t)) \wedge (\text{postd}(a) < k)\} \quad (3.3)$$

With respect to the example shown in Figure 3.3, we assume that the cache is an 1-way set associative cache (direct mapped cache). So, the frontier address set is $\{0, 2\}$, and the back address set is $\{3, 4\}$.

Regarding the back address set defined in Definition 4, the following property can be derived.

Lemma 1 In a cache with the LRU cache block replacement policy, the block accessed by an address of task t remains in the cache after the execution of t if and only if the

address is in back(t).

Proof: (1) Sufficiency: For an k -way set associative cache with the LRU replacement policy, a cache block in a set is replaced by an incoming new block if and only if the following conditions are valid: (1) the set has no empty blocks, (2) the new block has different block address from the k blocks in the set, and (3) the block to be replaced is the least recently used one in the set. The replacing order in the LRU policy exactly corresponds to the order of addresses occurring in the address sequence. If the block address of an address has the same cache mapping set as at most $(k-1)$ other distinct block addresses that follow the last occurrence of it, it can be easily verified that conditions (1), (2) and (3) cannot be valid at the same time. By this and the definition of back , the data block of every address in $\text{back}(t)$ must remain in the cache after the execution of t .

(2) Necessity: Assume that the block accessed by address b ($b \notin \text{back}(t)$) remains in the cache after the execution of t and the block does not contain any address in $\text{back}(t)$. By $b \notin \text{back}(t)$ and the definition of back , we know that either (a) $\text{postd}(b) = \infty$ or (b) $\text{postd}(b) \geq k$. Case (a): if $\text{postd}(b) = \infty$, b has the same cache mapping set as other k addresses that have distinct block addresses in the subsequent address sequence. By the LRU replacement policy, the block accessed by b must be replaced in the subsequent execution. This contradicts the assumption case. Case (b): if $\text{postd}(b) \geq k$, by the definition of postd , we can find another address c in the subsequent address sequence with the property that $\text{postd}(c) \neq \infty$ and c accesses the block of b . If $\text{postd}(c) \geq k$, the block of c , i.e., the block of b , cannot be in the cache after the execution of t . This contradicts the assumption case. If $\text{postd}(c) < k$, c must be in $\text{back}(t)$. This also contradicts the assumption case. So the Lemma 1 is valid.

□

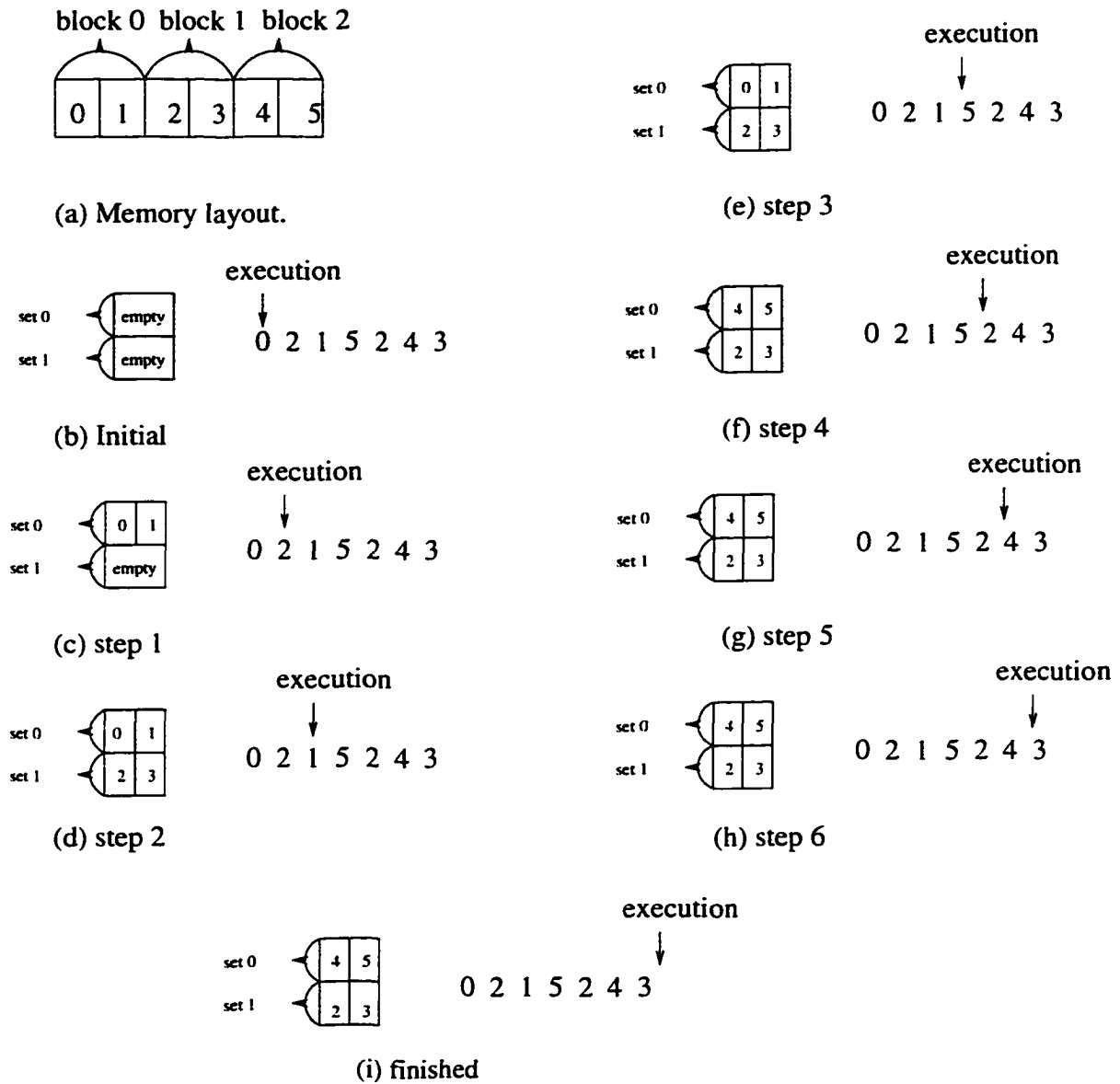


Figure 3.4: Exemplifying Lemma 1 for a direct-mapped two-set cache with a block size of 2.

In Figure 3.4, the cache-access pattern of the execution sequence in Figure 3.3 is shown. After execution, the cache contains blocks accessed by addresses in $\text{back}(t) = \{3, 4\}$. The following Corollary follows directly from Lemma 1 and the definition of postd .

Corollary 1 *After the execution of a task t , the data block accessed by each address a in $\text{back}(t)$ is the $(k - \text{postd}(a))$ -th least recently used block with respect to the other blocks in the same cache set.*

For the frontier address set defined in Definition 4, the following property holds.

Lemma 2 *Let a be an address of task t . If the block of a was most recently pre-loaded into a cache with respect to the other blocks in the cache set of a before the execution of t , the preloaded block will stay in the cache to be first accessed by task t at address a if and only if a is in $\text{frontier}(t)$.*

Proof: Let the cache be a k -way set associative cache. (1)Sufficiency: Because a is most recently pre-loaded into the cache with respect to the other blocks in the cache set of a , it will be replaced when and only when the other k distinct new blocks will be mapped into the same set (by the LRU replacement policy). If a is in $\text{frontier}(t)$, there are at most $k-1$ distinct new blocks that will be mapped into the cache set of a before the first access of task t at a . So, the preloaded block of a is still in the cache when t first accesses at a . Because $\text{pred}(a) \neq \infty$, there is no other distinct address accessing the preloaded block of a before a . So, a is the first address accessing the preloaded block.

(2)Necessity: Assume that a preloaded block stays in the cache to be first accessed by address a ($a \notin \text{frontier}(t)$). By $a \notin \text{frontier}(t)$ and the definition of frontier , either (a) $\text{pred}(a) = \infty$ or (b) $\text{pred}(a) \geq k$ is true. Case (a): if $\text{pred}(a) \geq k$, there are at least k distinct blocks that will be mapped into the cache set of a before the first occurrence of accessing to a . By LRU, the preloaded block of a must be

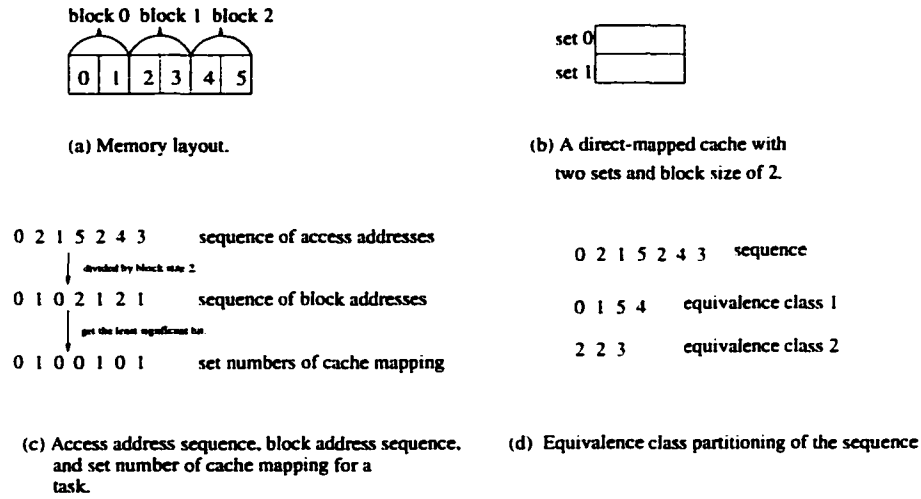


Figure 3.5: Exemplifying Lemma 2.

replaced before the first access to a . So, the first access of task t to a cannot be hit by the preloaded block. This contradicts the assumption. Case (b): if $\text{pred}(a) = \infty$, by the definition of pred we know that there is another distinct address b that occurs before the first occurrence of a and accesses the same block as a . So, a cannot be the first access to the preloaded block of a if it stays in the cache. This contradicts the assumption. So, Lemma 2 is true. \square

By Lemma 2, we know that only the memory accesses of task t at addresses in $\text{frontier}(t)$ are eliminated due to the reuse of the cache data preloaded before the execution of t . In the example given in Figure 3.3, we assume that the cache is a direct-mapped two-set cache with a block size of 2. If we consider accesses with the same cache mapping set be in an equivalence class, the original sequence can be classified into two subsequences, each in an equivalence class, as shown in Figure 3.5(d). The frontier address set of the sequence is $\{0, 2\}$, in which each address is the first access address in an equivalence class. For subsequence $0 \leftarrow 1 \leftarrow 5 \leftarrow 4$, only the memory access at 0 can be eliminated by reusing preloaded cache data. The access at 1 always hits no matter

what the cache is preloaded, the access at 5 always misses due to access 0. For another subsequence, lemma 2 can also be similarly verified.

Based on Lemma 1 and Lemma 2, we can obtain the following theorem for calculating the reusability of a sequential execution.

Theorem 1 *Given two tasks t_1 , t_2 , and an address $a \in \text{Dset}(t_2)$. During the consecutive execution of t_1 followed by t_2 on a k -way set associative cache, the access at address a of t_2 will reuse the cache data brought in by t_1 if and only if the following is true:*

$$(a \in \text{frontier}(t_2)) \wedge \exists b \in \text{back}(t_1)((a@b) \wedge (\text{postd}(b) + \text{pred}(a) < k)).$$

Proof: (a) Necessity: By Lemma 2 $a \in \text{frontier}(t_2)$ can be directly derived. Because the access at address a of t_2 will reuse the cache data brought in by t_1 , the block accessed by a must be loaded into cache by an access of t_1 at an address, denoted as b , and remains there after the execution of t_1 . By Lemma 1 and the definition of $@$, we have $(b \in \text{back}(t_1) \wedge (a@b))$. By Corollary 1 and the LRU replacement policy, the block accessed by b remains to be reused by the access of t_2 at a only when there are at most $(k - \text{postd}(b) - 1)$ distinct new blocks that will be mapped into the cache set of b before the first occurrence of a . By the definition of pred , $\text{pred}(a)$ gives the number of distinct blocks that will be mapped into the cache set of a before the first occurrence of a . By $a@b$, we have $(k - \text{postd}(b) - 1) \geq \text{pred}(a)$, namely, $\text{postd}(b) + \text{pred}(a) < k$. Moreover, sufficiency can be derived similarly to the necessity proof based on Lemma 1 and Lemma 2.

□

For a direct-mapped cache, Theorem 1 can be restated as the following Corollary.

Corollary 2 *Given two tasks t_1 , t_2 , and an address $a \in \text{Dset}(t_2)$. For a consecutive execution of t_1 followed by t_2 on a direct-mapped cache, the access to a of t_2 will reuse the cache data brought in by t_1 if and only if address a is cache-reusable with an address*

in $\text{back}(t_1)$ and is in $\text{frontier}(t_2)$, namely

$$\exists b \in \text{back}(t_1)(a@b) \wedge (a \in \text{frontier}(t_2)).$$

Based on Theorem 1, for any two sequential tasks t_1 and t_2 , the reusability of sequential execution $t_1 t_2$ on an k -way set associative cache ($k \geq 1$) is calculated as:

$$\text{Reuse}(t_1 t_2) = |\{a | (a \in \text{frontier}(t_2)) \wedge \exists b \in \text{back}(t_1)((a@b) \wedge (\text{postd}(b) + \text{pred}(a) \leq k))\}|.$$

So, for a sequential execution of n sequential tasks: t_1, t_2, \dots, t_n , we have

$$\text{Reuse}(t_1 t_2 \dots t_n) = \sum_{i=1}^{n-1} \text{Reuse}(t_i t_{i+1}).$$

On an uniprocessor, locality exploitation is aimed at finding an execution order of tasks with maximal data reusability. For data-independent tasks, this problem can be transferred into a graph search problem by representing tasks by graph nodes and the execution order from node t_i to node t_j by a directed edge with weight of $\text{Reuse}(t_i t_j)$. We call the derived graph as the Cache Data Reuse graph, denoted CDR graph. Hence, the optimal cache locality exploitation problem among k data independent tasks on an uniprocessor is equivalent to the problem of finding a simple directed path with maximal sum of edge weights in a weighted directed graph. This problem is equivalent to the famous traveling salesman problem, which is known to be an NP-complete problem.

The above formulation procedure presents an approach to find optimal locality optimization method on an uniprocessor. The question of whether there is a more efficient way to derive the optimal solution remains open. Based on our study, the analysis on memory access sequences is the foundation, which is only practical for a static compiler, not for a run-time system. As we pointed out before, the memory-access patterns of some applications are determined by run-time data, which is unpredictable at compile-time. So, for this type of application, searching an optimal solution is unrealistic even for uniprocessors.

3.3.2 Locality Optimization of Parallel Executions

From a practical point of view, minimizing the execution times of applications is the final goal for locality optimizations. On a uniprocessor, maximizing the reusability is consistent with this goal for a set of sequential tasks. However, on a multiprocessor, load imbalance is a major factor complicating the precise formulation of the locality optimization problem. Here, we present an approximate formulation to model the problem on multiprocessor systems.

In a cache coherent multiprocessor system with p processors, the cache locality problem is complicated by data-access interference among multiple caches, — a memory block is simultaneously accessed by multiprocessors. Previous research has shown that higher interference would cause more memory accesses. Because the interference is affected by the cache coherence protocol and the relative execution speeds of the processors, it is very difficult or impossible to quantify the degree of interference precisely. Here, we use an approximate metric, sharing degree, to quantify the interference. First, we define the sharing set as follows.

Definition 5 *Let P_1 and P_2 be two sets of tasks respectively executing on two different processors. The sharing set, denoted as $\text{sharing}(P_1, P_2)$, between P_1 and P_2 is defined as*

$$\begin{aligned} \text{sharing}(P_1, P_2) = \{a | (a \in Dset(P_1) \wedge \exists b \in Dset(P_2)(a@b)) \\ \vee (a \in Dset(P_2) \wedge \exists b \in Dset(P_1)(a@b))\}. \end{aligned} \quad (3.4)$$

where $Dset(P_1)$ and $Dset(P_2)$ represent the unions of the $Dsets$ of the tasks, respectively, in P_1 and P_2 .

Intuitively, $\text{sharing}(P_1, P_2)$ exactly gives the set of addresses where the corresponding memory blocks are accessed by both processors. In practice, each address in $\text{sharing}(P_1, P_2)$ is either a truly shared address accessible by both processors, or a false

shared address that is only accessed by one processor but resides on the same memory block as some accessed addresses of the other processors. The size of $sharing(P_1, P_2)$, called the sharing degree, is a quantitative measure of the cache coherent overhead in the parallel execution of P_1 and P_2 .

As an approximation, an optimal cache-locality exploitation problem of k data independent tasks on an SMP with p processors can be abstracted as the following p -partition problem:

1. Partitioning tasks into p parts: P_1, P_2, \dots, P_p , so that

- (a) Minimizing

$$\sum_{(1 \leq i, j \leq p) \wedge (i \neq j)} |sharing(P_i, P_j)|,$$

which aims at minimizing interference overhead; and

- (b) Minimizing the local imbalance among parts, namely minimizing

$$\sum_{i=1}^p (T_{max} - T_i),$$

where T_i is the execution time of part P_i ($i = 1, \dots, n$), and

$$T_{max} = \max\{T_i | i = 1, \dots, n\}.$$

2. Scheduling the sequential execution of tasks allocated on a processor, so that reusability is maximized. This has been modeled in Section 3.3.1.

Here, both problems 1 and 2 are NP-complete.

3.3.3 Implications of Models

Our locality optimization model indicates that a locality optimization method for parallel applications contains two major functions: task partitioning and task reordering.

The task partitioning function tries to achieve two conflict goals: to minimize data sharing and to provide balanced load among processors. Usually, tasks accessing larger sets of data tend to have more data sharing among them. Minimizing data sharing among partitions tends to put tasks accessing larger sets of data together. Because a task accessing a larger set of data usually has a larger computation granularity, the minimization of data sharing may result in imbalanced partitions. This would offset the benefit of minimizing data sharing. In theory, we may design an iterative method to find a convergence point that minimizes both data-sharing and load imbalance. This approach could hardly be applied in practice due to the high computational complexity of an iterative method. Because load imbalance is an important performance factor [5, 34, 73, 49, 47, 51, 61, 35, 83, 93, 94], we give load balance priority than data sharing. So, we simplify the task partitioning function as follows: *finding the balanced partitioning that has minimal data sharing*.

This simplification benefits the design of efficient partitioning algorithms. To achieve this partitioning goal, having information on task load and data sharing pattern among tasks is a necessary condition. Applications whose memory-access patterns are determined by run-time data usually have irregular computational patterns. As shown by research work in [49, 47, 51, 61, 35, 83, 93, 94], the load balance of a parallel application with an irregular computation pattern can only be achieved using run-time load balancing techniques. In addition, the minimization of data-sharing also relies on run-time analysis on memory-access patterns of applications.

In the task reordering function, the execution order of tasks on each processor is determined to maximize the data reuse in the cache. This requires the exploitation of cache-access patterns of tasks, which is determined by the data-access pattern and the underlying cache architecture. On physically addressed caches, this is further affected by the operating system. In complicated applications, these information can only be exploited at run-time.

Based on the above analysis, the locality optimization methods for applications with dynamic memory-access patterns must contain the following four components:

1. *A prediction method* to provide information on memory-access patterns of applications.
2. *A task partitioning algorithm* to dynamically partition parallel tasks of an application onto multiprocessors, aiming at minimizing both data sharing and load imbalance.
3. *A reordering algorithm* to determine the sequential execution order of the parallel tasks allocated on a processor, aiming at maximizing data reuse in the cache.
4. *A dynamic scheduling algorithm* to achieve guaranteed load balance, aiming at minimizing the total execution time.

As pointed out in the cache-locality optimization model, the optimal solutions for the last three components are NP-complete. Only heuristic solutions are feasible. In addition, because the execution overhead of a run-time method contributes directly to the execution time of a parallel application, a run-time method must be efficiently designed to prevent the benefit of optimizing cache locality from being nullified by run-time overhead. We use the well-known system design wisdom: *Simple is efficient*. This principle has led to the success of RISC computer systems [30].

Chapter 4

System Framework and Information Abstraction

In this chapter, we first present an overview of the functionality of the run-time system. Then, we introduce the information estimation technique and the internal representation of memory-access patterns. The programming interface is explained and illustrated by examples.

4.1 Run-Time System Framework

Our run-time system is implemented as a set of library functions which are called by a sequential application program at run-time. The run-time functions will create a set of parallel tasks to be executed in a SMP multiprocessor. Two major advantages of this approach are:

1. It carries compile-time information into the run-time phase to assist run-time analysis.

2. It uses run-time information on dynamic memory-access patterns to improve performance for a wide range of applications.

Although a run-time system can use all the static information analyzed at compile-time, it can not conduct as complicated an analysis as a compiler does because the run-time overhead directly affects application performance. Hence, in the design of our run-time system, we only carry forward a set of application dependent hints that can be efficiently used for locality optimization.

Figure 4.1 presents a framework for our run-time system. A given sequential application program is first transformed by a compiler or rewritten by a user to insert run-time functions. The generated executable file is encoded with application-dependent hints. At run-time, the encoded run-time functions are executed to fulfill the following functionalities:

1. *Estimation of the memory-access pattern.* Based on application dependent hints, a multidimensional memory space is built internally to represent the range of memory accesses for an application program. Meanwhile, the parallel tasks in an application are mapped onto the memory-access space based on their memory-access patterns. This abstract representation provides an important foundation for conducting locality optimization at run-time.
2. *Task grouping (or reordering).* Based on the distribution of tasks in the abstracted memory-access space, tasks are reorganized into groups using information on the underlying cache architecture. The tasks in a group are expected to heavily reuse their data in the cache.
3. *Task partitioning.* Based on the abstracted memory-access space, task groups are partitioned onto multiple task queues, each corresponding to a processor. Data sharing and load imbalance are minimized here.

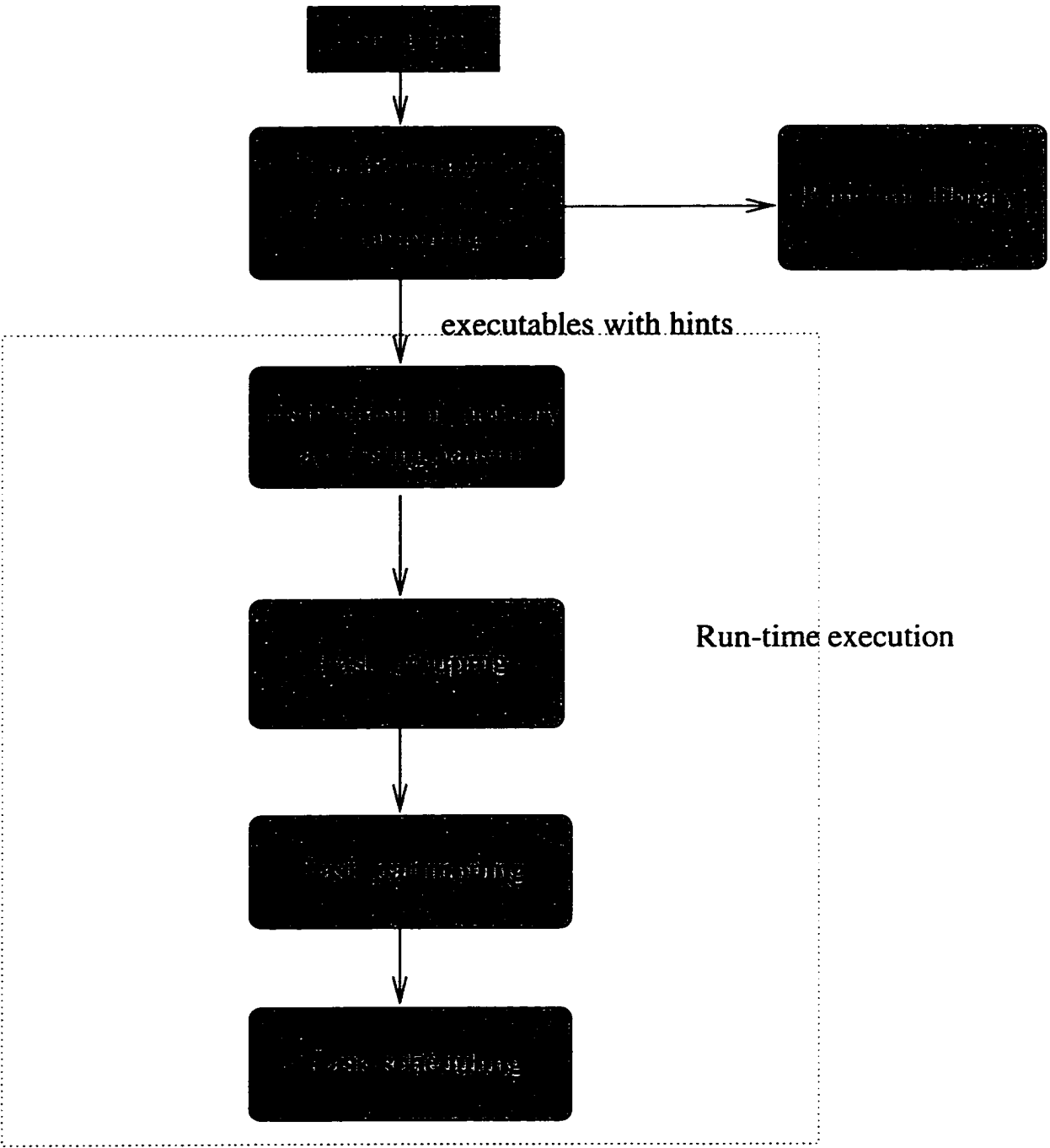


Figure 4.1: Execution framework of the run-time system.

4. *Task scheduling.* For application programs with irregular computation patterns or dynamic memory-access patterns, it is hard to achieve a balanced partitioning due to the lack of a precise prediction on task load. In addition, the run-time interference among multiprocessors is another potential factor causing imbalanced execution. To guarantee balanced execution, partitioned tasks are scheduled to execute in an adaptive way.

4.2 Information Estimation

4.2.1 Application Dependent Hints

The data referenced by the loop body of the nested loop described in section 3.1 has array structures, where each execution instance of the loop body accesses pieces of data separately in several arrays. Thus, estimating the access pattern of a loop on arrays is a major task.

Let A_1, A_2, \dots, A_n be n arrays accessed in the loop body of a nested data-independent loop. Each array is usually laid out in a contiguous memory region, independent of the other arrays. In rare cases, an array may be laid out across several uncontiguous memory pages. Although our run-time system may not handle these rare cases efficiently, the system works well for most memory layout cases in practice. Visualizing an array in an independent dimension, the memory regions of the n arrays can be integrally abstracted as an n -dimensional memory-access space, expressed as (A_1, A_2, \dots, A_n) . This n -dimensional memory-access space actually contains all the memory addresses that are accessed by a loop. In order for the run-time system to capture this memory space information precisely, the following three hints must be provided by the interface:

Hint 1: *The number of arrays, n .*

Hint 2: *The memory size of each array that is accessed by an application program.*

Hint 3: *The start physical address of the referenced memory address space of each array.*

Here **Hint 1** is static information. In **Hint 2**, the size is the distance between the address of the first referenced array element and the memory address right after the last referenced array element. This is estimated by the size of the virtual address space of an array, i.e., the product of the number of array elements in the reference space and the size of each array element in bytes. The array size may be static, where the size is known at compile-time, or dynamic, where the size is determined by run-time data. The hint should tell how to calculate the size at run-time. The run-time system maintains a Referenced Memory Space vector (s_1, s_2, \dots, s_n) , denoted the RMS vector, where $s_i (i = 1, 2, \dots, n)$ is the size of the referenced memory address space of array A_i . The RMS vector is used as an estimate of the total size of memory address space referenced by an application program.

From **Hint 3**, the underlying run-time system constructs a start address vector (b_1, b_2, \dots, b_n) , denoted as the SA vector, where $b_i (i = 1, 2, \dots, n)$ is the start address of the referenced memory address space of array A_i . The start addresses are dynamic, because memory addresses can only be determined at run-time. **Hint 3** tells how to determine the start addresses at run-time.

When a compiler or a user provides the above two hints, some calculation functions are formed for the run-time system to generate concrete information at run-time. The calculation functions are usually very simple. We will exemplify this in next section.

After determining the global memory-access space of a loop, we need to determine how each parallel iteration accesses the global memory-access space, so that we can reorganize them to improve memory performance. Here, we abstract each instance of a loop body of a parallel loop as a parallel task. The access region of a task in an array is simply represented by the start address of its access region. So, the following hint should be provided by interface functions.

Hint 4: *A memory-access vector of task t_j :*

$$(a_{j1}, a_{j2}, \dots, a_{jn})$$

where a_{ji} is the start address of the referenced region on the i -th array by t_j ($i = 1, 2, \dots, n$).

In some loop structures, a loop body may not contiguously access an array so that the access region may not be precisely abstracted by the start address. In these cases, more complicated estimation techniques should be used, such as adding multiple start addresses and ending addresses. However, this may result in unacceptable run-time overhead by significantly extending execution time for collecting more addresses and by making locality-oriented transformations more complex. This topic will be addressed in detail in Chapter 8. Here, we only attempt to use simple hints to do locality optimization.

Let $B(para_list)$ be a loop body function with parameter list $para_list$. In the run-time system, each instance of the loop body is created as a task $t_i(para_list, a_{i1}, a_{i2}, \dots, a_{in})$, where a_{ij} is the start address of the instance's access region on array A_j , for $j = 1, \dots, n$. In the following discussion, each task is simply represented as $t_i(a_{i1}, a_{i2}, \dots, a_{in})$.

In addition, the following hint should also be provided to assist task partitioning.

Hint 5: *The number of processors, p .*

By providing the above five hints, a compiler or a user does not need to conduct a complicated analysis. The locality optimization is handled in the program automatically. This eases the user's programming burden.

4.2.2 Abstract Representation of Memory-Accessing Space

Let A_1, A_2, \dots, A_n be the arrays accessed by a parallel loop. Let its RMS vector be (s_1, s_2, \dots, s_n) and its SA vector be (b_1, b_2, \dots, b_n) . The memory-access space of the loop

```

double A[X], B[Y], C[M][M];
int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y];

sparse-mm()
{
    int i, j, k, r, start, end;
    register double d;
    for (i=0; i<M; i++)
        for (j=0; j<M; j++){
            d = 0;
            start = Bcol[j]; end = Bcol[j+1];
            for (k=Arow[i]; k<Arow[i+1]; k++)
                for (r=start; r<end; r++)
                    if (Acol[k] == Brow[r]){
                        d += A[k]*B[r];
                        start = r+1;
                        break;
                    }
            C[i][j] = d;
        }
}

```

----> task $t(i, j)$

SMM: Sparse matrix multiplication.

Figure 4.2: Abstracting loop instances as tasks.

is abstracted as a n -dimensional memory-access space:

$$(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \dots, b_n : b_n + s_n - 1).$$

The total number of memory addresses contained is $\sum_{i=1}^n s_i$, not $\prod_{i=1}^n s_i$. Because the memory-access pattern of task t_i is estimated by a memory-access vector $(a_{i1}, a_{i2}, \dots, a_{in})$, the task t_i actually corresponds to point $(a_{i1}, a_{i2}, \dots, a_{in})$ in the n -dimensional memory-access space. This abstraction represents the memory-access pattern using a geometric model, which provides an effective base for the locality optimization at run-time.

We illustrate this idea by an example in Figure 4.2. This is a sparse matrix-matrix multiplication kernel, where the two outer loops are parallel loops and each instance of the loop body of loop j is abstracted as a task $t(i, j)$. Each task indirectly accesses a piece of array A and array B , respectively. The access patterns of each task on two arrays are determined by data in $Arow$ and $Bcol$, which are hard to analyze at compile-time. However, the start access addresses of task $t(i, j)$ on A and B can be obtained at run-time by executing $\&A[Arow[i]]$ and $\&B[Bcol[j]]$ respectively.

```
double B[100], A[200];
```

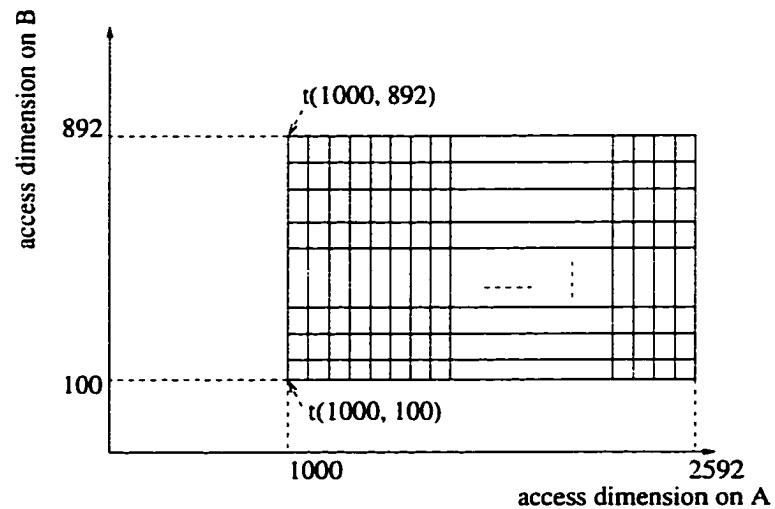
Memory layout of A : size = 200×8 ; starting at $\&A[0] = 1000$:

Memory layout of B : size = 100×8 ; starting at $\&B[0] = 100$:

(a) hints on memory layouts of two accessed arrays.

100	108	116	124	892	900	1000	1008	1016	2592	2600
B[0]	B[1]	B[2]	----	B[100]	-----	A[0]	A[1]	A[2]	-----	A[200]

(b) Physical memory layout



(c) An 2-dimensional memory-accessing space.

Figure 4.3: An abstract representation.

Figure 4.3 presents an example of the abstract representation of the memory accesses based on the physical memory layout of arrays A and B only. Figure 4.3(a) gives the hints on the memory-access space. Figure 4.3(b) illustrates the memory layout of two arrays where B and A are laid out at start address 100 and 1000 respectively. Each array element has size of 8 bytes. The memory space of arrays A and B is the whole memory space accessed by tasks. Then, the memory-access space is represented as a 2-dimensional space as shown in Figure 4.3(c), where each point gives a pair of possible start memory-access addresses on A and B respectively by a task. For example, $t(1000, 100)$ means task t will access array A at start memory address 1000, and access array B at start physical address 100. In the next chapter, we explain how these application-dependent hints are used to assist the exploitation of cache locality of applications based on the memory-access space.

4.3 Interface and Programming Example

4.3.1 Application Programming Interface (API)

The interface functions are mainly used to provide application-dependent hints for the run-time system. The current system is implemented in C language. The API of the system is very simple, and consists of the following three functions:

1. `void cacheminer_init(int csize, float f, int p, int n, long s1, void * b1, ..., long sn, void * bn)`

This function provides the following types of hints:

- Cache size `csize` gives the size in KB of the secondary cache on each processor.
- Task control parameter `f` is a number in $(0, 1]$, giving the usage percentage of a cache to cluster tasks. (Detailed descriptions will be presented in a later

chapter.)

- p is the number of processors.
- n is the total number of arrays, on which hints are provided.
- Hints on arrays are given in pairs. Each pair of s_j and b_j ($j = 1, 2, \dots, n$), give respectively the size and start physical address of a referenced array by tasks. All the arrays are arranged in a user-defined order. (There is no specific requirement on the array order.)

Based on the information provided by this function, the run-time system builds a $(n + 1)$ -dimensional hash structure. (the hash structure will be discussed in the next chapters and is transparent to users.)

2. `task_create(void (* f), int m, int t_1, \dots, t_m , void $*a_1, \dots, *a_m$)`

This function creates a task with its computing function, denoted as `void f(t_1, t_2, \dots, t_m)`, and carries hints a_1, a_2, \dots, a_m on the access pattern of the task into the run-time system. Here a_i ($i=1, \dots, m$) is the start access address of the task on i -th array (arrays are pre-ordered by a user).

If the number of hints, m , is larger than the number of hinted data arrays, n , only the first n hints are used. This flexibility would allow a task function to have a larger number of parameters than that of the accessed data arrays. However, m cannot be smaller than n , which is easily achieved in programming by using dummy parameters in a task function. The order of hints here must be the same as that of those hints presented in `cacheminer_init`, i.e., a_j, s_j , and b_j are hints on the same array for $j = 1, 2, \dots, n$.

3. `void task_run(int repeat)`

Computation pattern	Memory-access pattern	Data-dependence
regular	static	static
regular	static	dynamic
regular	dynamic	static
regular	dynamic	dynamic
irregular	static	static
irregular	static	dynamic
irregular	dynamic	static
irregular	dynamic	dynamic

Table 4.1: Classification of applications.

This function starts the run-time system to execute the tasks in the hash structure in parallel. If the tasks are going to execute at second time, the variable `repeat` is set to 1 so that the run-time system can keep the hash structure in order to eliminate the overhead of rebuilding it. In this situation, the run-time system exploits processor affinity by using old partitions of tasks. Otherwise, the variable `repeat` is set to 0.

4.3.2 Classification and Programming of Applications

In order to reflect all types of applications while not getting into exhaustive investigation, we classify applications based on three factors: computation pattern, memory-access pattern, and data-dependence pattern. Computation patterns can be classified as two types: regular, where the computation tasks of an application are naturally balanced, and irregular, where the computation tasks of an application are not naturally balanced.

Furthermore, memory-access patterns and data-dependence patterns can be respectively classified into static patterns that are determinable at compile-time and dynamic patterns, which are not determinable at compile-time. Intuitively, based on these patterns, applications can be classified into eight types as shown in Table 4.1. However, the computation pattern, the memory-access pattern, and the data-dependence pattern interact one another. Usually, when an application has a dynamic memory-access pattern or a dynamic data-dependence pattern, it has an irregular computation pattern. In addition, the memory-access pattern of an application is affected by its data-dependence pattern. When the data-dependence pattern is not determinable at compile-time, the memory-access pattern must be not determinable. Considering these effects, applications finally fall into the following four types, listed in increasing difficulty degree for locality optimization.

Type 1 Applications with regular computation patterns, static memory-access patterns, and static data dependence.

Type 2 Applications with irregular computation patterns, static memory-access patterns, and static data dependence.

Type 3 Applications with irregular computation patterns, dynamic memory-access patterns, and static data dependence.

Type 4 Applications with irregular computation patterns, dynamic memory-access patterns, and dynamic data dependence.

Based on the above classification, we choose each benchmark from the first three application types that fit into our programming model. For the most difficult applications in type 4, our technique can also be used to improve memory performance by combining some data-dependence detection techniques. Because data-dependence detection is not

```

double C[N][N], A[N][N], B[N][N];

dense_mm()
{
    int i, j, k;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            for (k=0; k<N; k++)
                C[i][j] += A[i][k]*B[j][k];
}

double C[N][N], A[N][N], B[N][N];

dmm_rt(float f, int p)
{
    int i, j, k;
    long s = N*N*sizeof(double);
    cacheminer_init(Csize, f, p, 2, s, a[0], s, b[0]);
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            task_create(fun, 2, i, j, A[i], B[j]);
    task_run(0);
}

fun(int i, int j)
{
    int k;
    for (k=0; k<N; k++)
        C[i][j] += A[i][k]*B[j][k];
}

```

Figure 4.4: Dense matrix multiplication: the sequential program is given on the left and the locality optimized version on the run-time system is given on the right.

in the focus of this dissertation, we discuss this in Chapter 8. Here, based on selected applications, we show how to rewrite ordinary C programs through the run-time system to exploit different types of cache localities in a SMP system. These applications are used to evaluate the run-time system.

An Example of Type 1: Dense Matrix Multiplication

The sequential program of a dense matrix multiplication (DMM) is given on the left side in Figure 4.4. To increase locality, array B has been transposed so that its access in loop k is consistent with the memory layout of B to increase spatial locality. Loop i and loop j are two parallel loops. To optimize the cache locality of the DMM, the innermost loop, i.e., loop k, is considered as a task with two input parameters: i and j, denoted as $t(i, j)$.

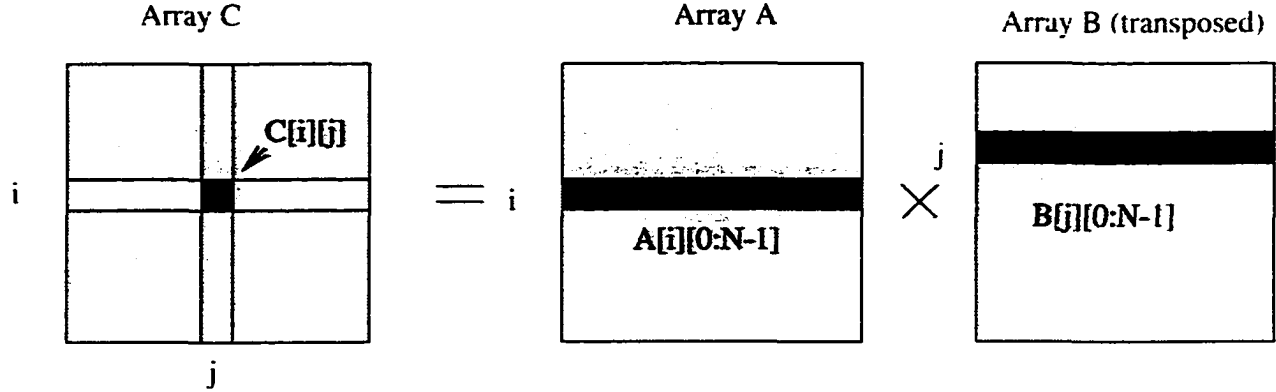


Figure 4.5: Regular computation pattern in the DMM.

Task $t(i, j)$ calculates $C[i][j]$ using a row of A that starts at address $A[i]$ and a row of B that starts at address $B[j]$. The memory-access pattern of task $t(i, j)$ is estimated by $(A[i], B[j])$. The computation pattern is illustrated in Figure 4.5, which is highly regular.

With the support of the run-time library functions, the original sequential program of the DMM is rewritten and listed on the right side in Figure 4.4. In function `dmm_rt`, the two outer loops create N^2 parallel tasks with the same computation load. Function `task_run` starts the parallel execution of tasks.

An Example of Type 2: Adjoint Convolution

An adjoint convolution (AC) kernel is given on the left side in Figure 4.6. The data-dependence is carried by the inner loop. The outer loop i is a parallel loop. Considering each instance of the inner loop j as a task, the outer loop will create N^2 parallel tasks: $\{t(i, x) \mid i = 0, \dots, N^2\}$. The task $t(i, x)$ accesses $(N^2 - i)$ elements of B and C respectively, starting at $\&B[i]$ and $\&C[0]$. This memory-access pattern is static. But these tasks have an irregular computation pattern shown in Figure 4.7. The computation size of the tasks decreases as index i increases. So, the load imbalance in the AC is an

<pre> int A[N*N], B[N*N], C[N*N]; ac(int x) { int i, j; for (i=0; i<N*N; i++) for (j=i; j<N*N; j++) A[i] += x*B[j]*C[j-i]; } </pre>	<pre> int A[N*N], B[N*N], C[N*N]; ac(int x, float f, int p) { int i; long s = N*N*sizeof(double); cacheminer_init(Csize, f, p, 2, s, &B[0], s, &C[0]); for (i=0; i<N*N; i++) task_create2fun, 4, i, x, &B[i], &C[0]); task_run(0); } fun(int i, int x) { int j; for (j=i; j<N*N; j++) A[i] += x*B[j]*C[j-i]; } </pre>
--	---

Figure 4.6: Adjoint Convolution (AC): the sequential program is given on the left and the parallelized version on the Cacheminer system is given on the right.

important factor that should be carefully traded off with locality optimization.

In the transformation of the sequential AC program by the run-time system, no additional effort needs to be made for a user to take care of the load imbalance. The locality optimized version is shown on the right of Figure 4.6, which is similar to the optimized DMM. The load imbalance is taken care by the run-time system at the scheduling phase. However, it is more difficult for a compiler to exploit this application than for DMM.

An Example of Type 3: Sparse Matrix Multiplication

A sequential Sparse Matrix Multiplication (SMM) program is presented on the left side in Figure 4.8. In real systems, a sparse matrix usually has only 10% to 30% non-zero elements. In order to use the memory space efficiently for a sparse matrix, the matrix is generally represented in a dense form. Array elements stored in dense format can be

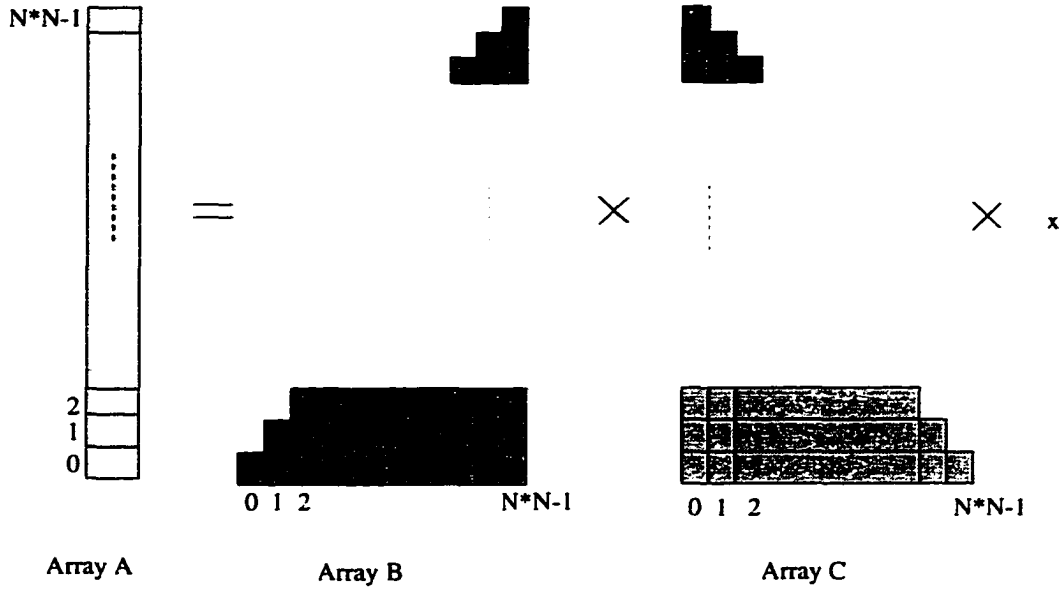


Figure 4.7: Irregular computation pattern in the AC.

indirectly accessed through auxiliary indexing arrays. Although this representation is memory efficient, it makes compiler analysis very difficult. Analysis is impossible when data is input at run-time.

In the SMM, non-zero elements of two $M \times M$ source sparse matrices are represented in two dense 1-dimensional arrays: A and B, respectively. The non-zero elements in A are stored by rows and the non-zero elements in B are stored by columns. Auxiliary array **Arow** gives the position in A of the first non-zero element of each row for the first source matrix. Auxiliary array **Acol** gives the column number of each non-zero element in A. Similarly, array **Bcol** gives the position in B of the first non-zero element of each column for the second source matrix. Array **Brow** gives the row number of each element in B. The sequential sparse matrix multiplication is described on the left side in Figure 4.8.

The two outer loops are data independent and are rewritten on the run-time system as described by the program presented on the right side in Figure 4.8. At

<pre> double A[X], B[Y], C[M][M]; int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y]; sparse_mm() { int i, j, k, r, start, end; for (i=0; i<M; i++) for (j=0; j<M; j++){ start = Bcol[j]; end = Bcol[j+1]; for (k=Arow[i]; k<Arow[i+1]; k++) for (r=start; r<end; r++) if (Acol[k] == Brow[r]){ C[i,j] += A[k]*B[r]; start = r + 1; break; } } } </pre>	<pre> double A[X], B[Y], C[M][M]; int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y]; sparse_mm(float f, int p) { int i, j, s = sizeof(double); cacheminer_init(Csize, f, p, 2, X*s, &A[0], Y*s, &B[0]); for (i=0; i<M; i++) for (j=0; j<M; j++) task_create(fun, 2, i, j, &A[Arow[i]], &B[Bcol[j]]); task_run(0); } fun(int i, int j) { int k, r, start, end; start = Bcol[j]; end = Bcol[j+1]; for (k=Arow[i]; k<Arow[i+1]; k++) for (r=start; r<end; r++) if (Acol[k] == Brow[r]){ C[i][j] += A[k]*B[r]; start = r + 1; break; } } } </pre>
--	---

Figure 4.8: Sparse matrix-matrix multiplication: the left side is the sequential program version and the right side is the rewritten version on the run-time system.

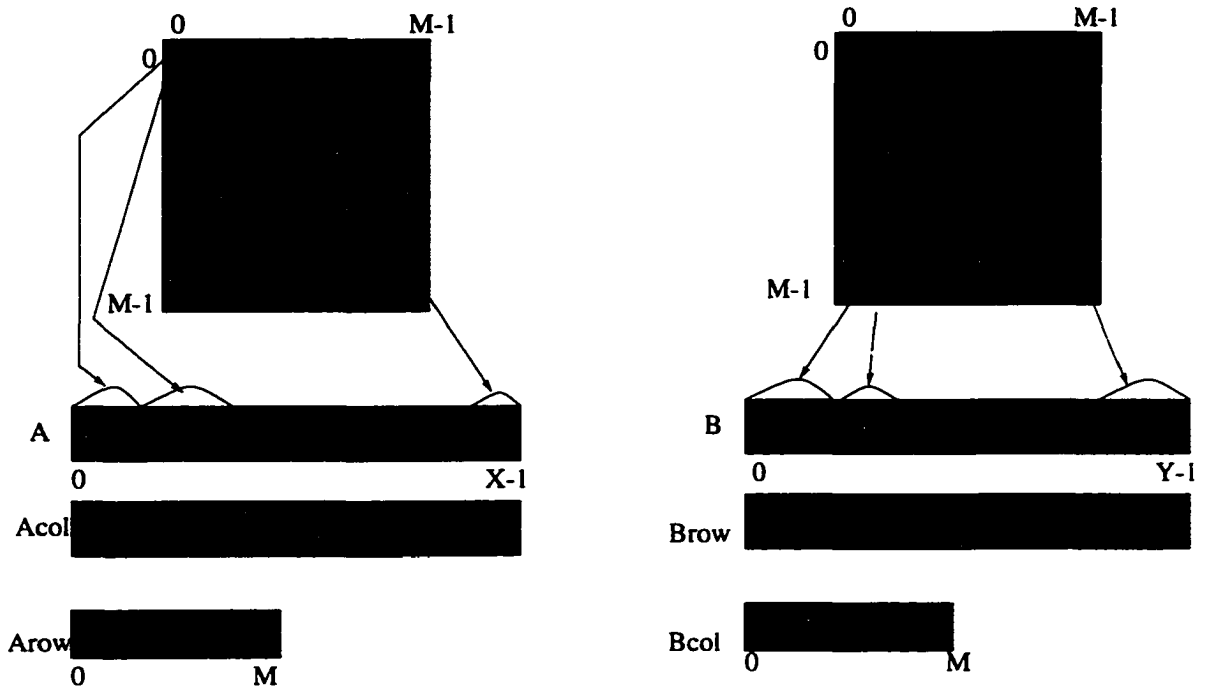


Figure 4.9: The dense representation in the SMM.

compile-time, it is known that all the elements in A and B will be accessed. In addition, we can know that task $t(i, j)$ accesses A and B at start addresses $\&A[Arow[i]]$ and $\&B[Bcol[j]]$, which can only be obtained at run-time.

The created tasks have irregular computation patterns. The tasks have an imbalanced workload determined by the input matrices. Figure 4.9 illustrates the dense representation whose memory-access patterns can only be determined at run-time, because the elements of arrays A and B accessed by each task are dependent on the data in auxiliary arrays. This application not only has a dynamic memory-access pattern, it also has an irregular computation pattern. It is hard for a compiler to exploit cache locality. But at run-time, the run-time system can use both static information and run-time information to conduct locality optimization.

Chapter 5

Memory-Layout Oriented Optimizations

The central functions of our run-time locality optimization system are task reordering and task partitioning in order to maximize data reuse in a partition and to minimize data sharing among balanced partitions. Optimal solutions of task reordering and partitioning are NP-complete, as shown in Section 3.3. So far, only reference [60] has addressed the run-time task reordering problem with the objective of improving the memory performance of sequential programs. In this chapter, we propose a more effective method based on the abstract representation given in Section 4.2.2. Run-time task partitioning with the objective of minimizing data-sharing and load imbalance is a more challenging problem, and has not been addressed before. This chapter proposes a heuristic algorithm to solve this problem.

5.1 Goals

At run-time, the estimated memory-access pattern of each task is determined. The goal of efficiently executing the tasks for cache locality on a SMP system is accomplished by reorganizing the tasks to maximize the cache data reuse in each processor, and by partitioning and mapping the reorganized tasks to multiprocessors to minimize the data sharing among processors.

From the standpoint of implementation, the basic idea of the task reorganization is to use an $(n+1)$ -dimensional hash structure to integrate the grouping and partitioning of the tasks with n hints. Let $(d_1, d_2, \dots, d_{n+1})$ be the $(n+1)$ -dimensional hash structure where $d_i (i = 1, 2, \dots, n+1)$ is the size of i -th dimension of the hash structure. A task bin is associated with each point in the $(n+1)$ -dimensional hash structure. Most importantly, the $(n+1)$ -dimensional hash structure can be considered to include d_{n+1} n -dimensional partitions: $(0 : d_1 - 1, 0 : d_2 - 1, \dots, 0 : d_n - 1, 0)$, $(0 : d_1 - 1, 0 : d_2 - 1, \dots, 0 : d_n - 1, 1)$, \dots , $(0 : d_1 - 1, 0 : d_2 - 1, \dots, 0 : d_n - 1, d_{n+1} - 1)$. The hash structure satisfies the following requirements:

1. *Each task can be efficiently mapped into an appropriate task bin in an n -dimensional partition by a set of hash functions.*
2. *All the tasks in a task bin have an opportunity to reuse their data in the cache. (An optimal solution is NP-complete.)*
3. *The data sharing between two different partitions is insignificant. (An optimal solution is NP-complete.)*
4. *All partitions are estimated to contain balanced numbers of tasks.*

In the following, we use memory-access space oriented shrinking and partitioning methods to find a hash structure and a set of hash functions so that tasks can be organized

in a way that has the above four characteristics.

5.2 Memory-access space shrinking

In our run-time system, for any given parallel loop, the memory-access pattern of its parallel tasks is captured by a multi-dimensional memory-access space in Section 4.2.2. Assume that $\{t_i(a_{i1}, a_{i2}, \dots, a_{in}) | i = 1, 2, \dots, m\}$ is a set of m parallel tasks in a given parallel loop created with n hints on n arrays. The accessed arrays are assumed to be organized in some order. This order can be any order based on user interests, but it must be consistent with the order used in interface functions. Based on the hints provided in interface functions, the run-time system obtains the following information about the parallel loop:

- RMS vector, (s_1, s_2, \dots, s_n) , where s_i ($i=1, \dots, n$) is the size in bytes of i -th array.
- RA vector, (b_1, b_2, \dots, b_n) , where b_i ($i=1, \dots, n$) is starting memory address of i -th array.
- p , the number of processors, C , the capacity of the underlying secondary cache in bytes, and n , the number of arrays accessed by parallel tasks.

The memory-access space of the parallel loop is $(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \dots, b_n : b_n + s_n - 1)$, where the i -th dimension is the referenced memory address space, $(b_i : b_i + s_i - 1)$, of i -th array. Conceptually, task t_i ($i=1, \dots, n$) is mapped onto point $(a_{i1}, a_{i2}, \dots, a_{in})$ in the memory-access space based on the starting memory addresses of their memory-access regions. So, the closer the task points are in the memory-access space, the more physical data these tasks will share. Hence, grouping nearby task points in the memory-access space to execute together has a good chance to enhance temporal locality and spatial locality. This is achieved by shrinking the memory-access space based on the underlying cache size in the following two steps.

In the first step, memory-access space $(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \dots, b_n : b_n + s_n - 1)$ is shifted into origin point $(0, \dots, 0)$ by subtracting (b_1, b_2, \dots, b_n) from the coordinates of all task points. The original n -dimensional memory-access space is transformed into the space $(0 : s_1 - 1, 0 : s_2 - 1, \dots, 0 : s_n - 1)$, called the transformed memory-access space. The transformation function is

$$f_1(a_{i1}, a_{i2}, \dots, a_{in}) = (a_{i1} - b_1, a_{i2} - b_2, \dots, a_{in} - b_n) \quad (5.1)$$

We know that nearby tasks in the memory-access space have great potential to reuse their data in caches. But, grouping nearby tasks optimally at run-time is a hard problem, because we cannot conduct necessary analyses to obtain complete information on the memory-access patterns of tasks. In this case, the best we can do is to assume that each task accesses each dimension of the memory-access space in the same pattern.

In the second step, we use a uniform blocking technique to evenly partition each dimension of the transformed memory-access space into segments. This will partition the transformed n -dimensional memory-access space into many polyhedrons with equal volume. The size of a partition refers to the volume of its corresponding polyhedron. Then, the tasks in a polyhedron are grouped together to execute. In order to minimize conflict misses in a group, the number of memory addresses accessed by the tasks in the polyhedron must be smaller than the underlying cache size. Based on this, we use fC/n as a partitioning size on each dimension, where $f (\leq 1)$ is a weight constant and C is the cache size. This actually results in polyhedrons which have equal size fC/n in every dimension. The total number of addresses covered in a polyhedron is fC , which is not larger than the cache size.

So, the tasks in a polyhedron, called a task reuse group hereafter, are estimated to have a better chance to reuse cache data when they execute consecutively. The internal execution order in a task reuse group is not important, because their accessed data is estimated to be completely held in a cache. Because each task reuse group has the

same size, it can be approximately considered to contain the same number of tasks. For applications with irregular memory-access patterns, this may not be true. This irregularity will be further handled in dynamic scheduling.

Here, f is an important parameter for controlling the number of tasks in a task reuse group. A larger f tends to allow more tasks to reuse the cache data, but may cause more interference. An optimal value of f may be difficult to predict at run-time, because this requires a precise analysis of the interference among the memory-access patterns of the tasks. We will evaluate the effect of interference on performance by decreasing f exponentially (e.g., using $1/2^x$, for $x = 0, 1, 2, 3$). When $f = 1/2^x$, the tasks are grouped in the way that all the tasks in a group only use approximately $1/2^x$ of the cache.

To group the tasks in each task reuse group (or polyhedron), each dimension of the transformed n -dimensional memory-access space, $(0 : s_1 - 1, 0 : s_2 - 1, \dots, 0 : s_n - 1)$, is shrunk by n/fC . This results in a new n -dimensional space, $(0 : \lfloor \frac{s_1-1}{fC/n} \rfloor, 0 : \lfloor \frac{s_2-1}{fC/n} \rfloor, \dots, 0 : \lfloor \frac{s_n-1}{fC/n} \rfloor)$, called the n -dimensional bin space. In the bin space, each point is associated with a task bin that holds all the tasks in the corresponding task reuse group in the transformed memory-access space. So, the shrinking function for a point $(a_{i1}, a_{i2}, \dots, a_{in})$ in the transformed memory-access space to be mapped onto the bin space is

$$f_2(a_{i1}, a_{i2}, \dots, a_{in}) = (\lfloor \frac{a_{i1}}{fC/n} \rfloor, \lfloor \frac{a_{i2}}{fC/n} \rfloor, \dots, \lfloor \frac{a_{in}}{fC/n} \rfloor). \quad (5.2)$$

In Figure 5.1, the shrinking procedure of the memory-access space is exemplified by the 2-dimensional memory-access space given in Figure 4.9. Before shrinking, the original memory-access space is shifted to origin point $(0,0)$ (see Figure 5.1(b)). The shifting function is shown in Figure 5.1(b). Then each dimension of the shifted memory-access space is shrunk by $C/2$ into a new 2-dimensional bin space in Figure 5.1(c). The tasks in the shadow square in Figure 5.1(b) would not access more space than the cache size, and are mapped onto one point in the bin space so that they can be grouped together to execute.

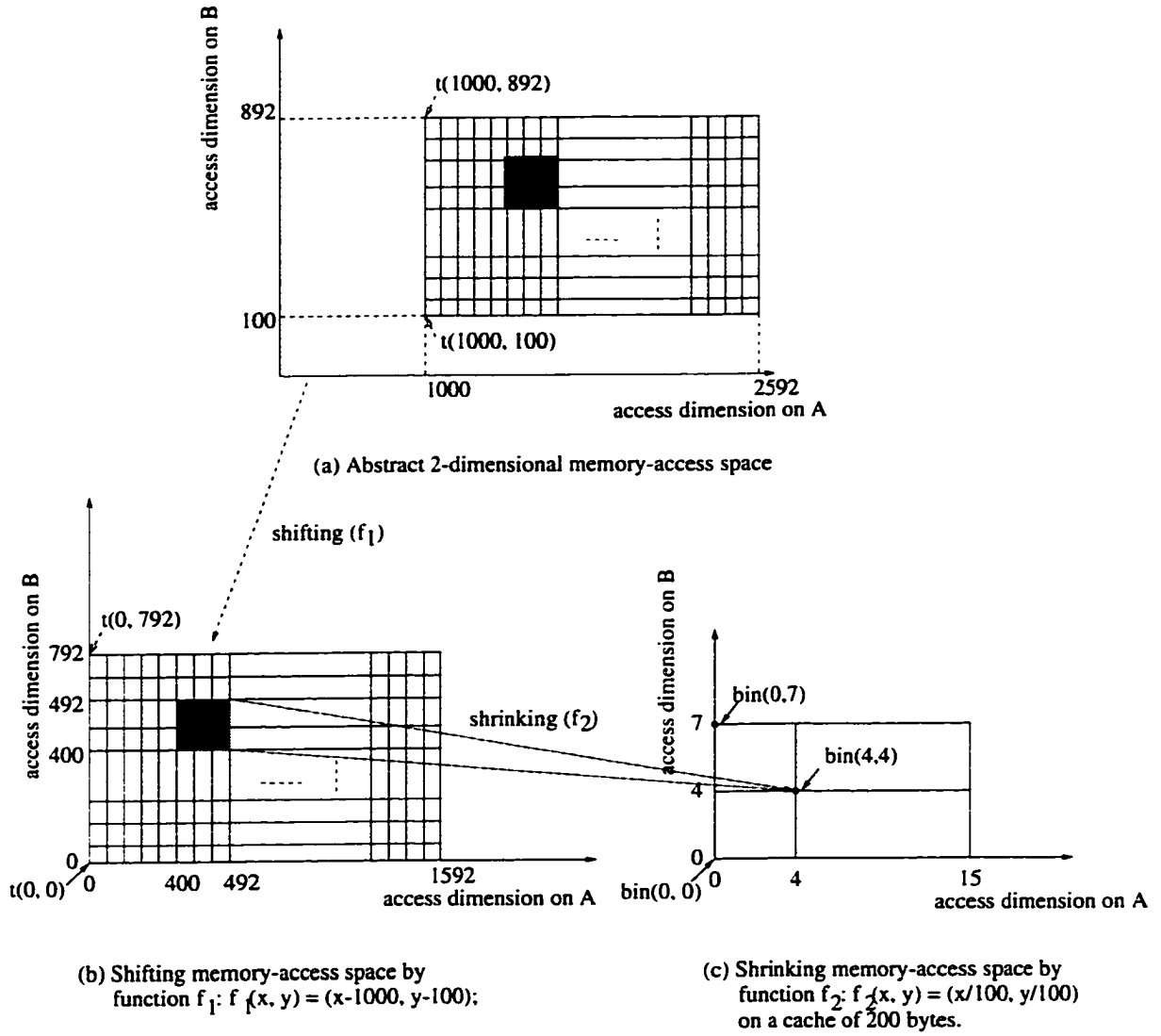


Figure 5.1: Memory-access space shrinking.

5.3 Bin Space Based Task Partitioning

5.3.1 Partition problem

After shrinking an n -dimensional memory-access space, tasks have been grouped based on locality affinity information in an n -dimensional bin space. Task partitioning is aimed at partitioning the n -dimensional bin space into p partitions (p is the number of processors and each partition is an n -dimensional polyhedron) so that

1. *the data sharing among partitions is minimized.*
2. *p partitions are balanced.*

For a given n -dimensional bin space, denoted $B^n(0:L_1, 0:L_2, \dots, 0:L_n)$ where L_i ($i=1, 2, \dots, n$) is the size of the i -th dimension, a partitioning method can be represented as a partitioning vector $\vec{k}(k_1, k_2, \dots, k_n)$ where the i -th dimension of the bin space is partitioned into k_i parts for $i = 1, 2, \dots, n$. A partitioning vector is formally defined as follows.

Definition 6 *With respect to an n -dimensional bin space B^n and p processors, an n -dimensional vector (k_1, k_2, \dots, k_n) is said to be a partition vector on B^n with respect to p processors if and only if*

$$\prod_{i=1}^n k_i = p.$$

So, a partitioning algorithm should consist of two functions:

1. Determining an optimal partitioning vector $\vec{k}(k_1, k_2, \dots, k_n)$ that satisfies the above two conditions.
2. Partitioning a bin space based on a selected partitioning vector.

To solve the above problems, we first need to specify the two partitioning conditions formally.

A good balance refers to a small deviation of execution times of partitions. Because it is hard at run-time to know task distribution in a bin space and load distribution in tasks, the execution times of partitions are unknown at run-time. In this case, we can only assume that tasks are uniformly distributed and that tasks have a balanced load. Obviously, this assumption is too restrictive for some applications. However, the imbalance possibly caused by this approximate assumption will be eliminated by run-time scheduling in the execution stage (in the next chapter). Based on this, we consider partitions balanced when they have equal size, which can be easily achieved by evenly partitioning each dimension of a bin space for a given partitioning vector. Hereafter, an even partitioning method is always used for a determined partitioning vector.

The major difficulty of designing a partitioning algorithm comes from the minimization requirement on data-sharing. First we need a metric to quantify the data-sharing. In Definition 5, *sharing* metric has been defined to quantify data sharing between two tasks. However, the calculation of *sharing* is based on memory-access address sequences of tasks, which is too expensive to obtain at run-time. Here, we use a more practical metric that may be less precise than the metric *sharing*, but much cheaper at run-time.

In data-independent loop structures, true data sharing comes only from reads, which will not cause cache coherence overhead in a shared-memory system. Here, only the false sharing will cause cache coherence overhead. When an n -dimensional bin space is partitioned, the most likely places for false sharing occur among adjacent boundary task bins between two partitions (or two n -dimensional polyhedrons). Heuristically, the data-sharing degree between two partitions can be determined by the volume of their boundary space (i.e., the intersection space between the two polyhedrons). The larger the boundary space between two partitions, the more they tend to have data-sharing.

So, an optimal partitioning method should minimize the sum of the volumes of boundary spaces among partitions.

If partitions are n -dimensional polyhedrons, the boundary space between any two partitions is an $(n - 1)$ -dimensional polyhedron, e.g., a cube for $n = 4$, a plane for $n = 3$, or a line for $n = 2$. Let $\rho(P_i, P_j)$ be the volume of the boundary space between two partitions P_i and P_j . (Here, $\rho(P_i, P_j) = \rho(P_j, P_i)$.) For a given partitioning vector \vec{k} and a bin space B^n , if r partitions, P_1, P_2, \dots , and P_r , are generated, the data-sharing degree, denoted $f_{sharing}(\vec{k}, B^n)$, of the partitions is estimated as

$$f_{sharing}(\vec{k}, B^n) = \sum_{1 \leq i < j \leq r} \rho(P_i, P_j).$$

So, an optimal partitioning vector (or method) should have a minimal data-sharing degree.

Because a bin space is orthogonal, i.e., all faces of the bin space are mutually orthogonal, all partitions obtained from the bin space are also orthogonal. It is easy to see that the boundary space of two partitions (if they have one) is an orthogonal polyhedron. Based on this, $f_{sharing}$ actually can be calculated by the following theorem.

Theorem 2 *If $B^n(0 : L_1, 0 : L_2, \dots, 0 : L_n)$ is an n -dimensional bin space that is partitioned by a partitioning vector $\vec{k}(k_1, k_2, \dots, k_n)$, then the sharing degree, $f_{sharing}(\vec{k}, B^n)$, of the resulted partitions is*

$$f_{sharing}(\vec{k}, B^n) = \sum_{i=1}^n ((k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j). \quad (5.3)$$

Proof: Let r be the number of elements in \vec{k} that do not equal 1. The proof is based on the reduction on r as follows.

1. When $r = 0$: The partitioning vector \vec{k} is $(1, 1, \dots, 1)$ which only produces one partition. So, the sharing degree among partitions should be 0. From equation (5.3), it can be verified that $f_{sharing}(\vec{k}, B^n) = 0$.

2. When $r \leq (n - 1)$: The theorem is assumed to be valid.
3. When $r = n$: Partitioning bin space $B^n(0 : L_1, 0 : L_2, \dots, 0 : L_n)$ using partitioning vector $\vec{k}(k_1, k_2, \dots, k_n)$ is equivalent to the following two partitioning steps:

Step 1 : Evenly partitioning bin space $B^n(0 : L_1, 0 : L_2, \dots, 0 : L_n)$ using partitioning vector $\vec{k}^1(1, 1, \dots, 1, k_n)$ into k_n subspaces $B_1^n(0 : L_1, 0 : L_2, \dots, 0 : L_{n-1}, 0 : L^1)$, $B_2^n(0 : L_1, 0 : L_2, \dots, 0 : L_{n-1}, 0 : L^2)$, \dots , $B_{k_n}^n(0 : L_1, 0 : L_2, \dots, 0 : L_{n-1}, 0 : L^{k_n})$. Let $r = L_n - \lfloor \frac{L_n}{k_n} \rfloor \times k_n$. The evenly partitioning method has the following properties:

$$L_i = \lfloor \frac{L_n}{k_n} \rfloor + 1, \text{ when } i \leq r; \quad (5.4)$$

$$L_i = \lfloor \frac{L_n}{k_n} \rfloor, \text{ when } i > r; \quad (5.5)$$

$$\sum_{i=1}^{k_n} L^i = L_n. \quad (5.6)$$

Based on the assumption, the sharing degree resulted from this step is

$$f_{sharing}(\vec{k}^1, B^n) = (k_n - 1) \prod_{j=1}^{n-1} L_j. \quad (5.7)$$

Step 2 : Using partitioning vector $\vec{k}^2(k_1, k_2, \dots, k_{n-1}, 1)$ to partition each subspace $B_i^n(0 : L_1, 0 : L_2, \dots, 0 : L_{n-1}, 0 : L^i)$ for $i = 1, 2, \dots, k_n$. Based on the assumption, the sharing degree of the partitions in subspace B_i^n is

$$f_{sharing}(\vec{k}^2, B_i^n) = \sum_{l=1}^{n-1} ((k_l - 1) \times \prod_{j=1 \wedge j \neq l}^{n-1} L_j) \times L^i. \quad (5.8)$$

Based on equations (5.7) and (5.8), we have

$$\begin{aligned} f_{sharing}(\vec{k}, B^n) &= f_{sharing}(\vec{k}^1, B^n) + \sum_{i=1}^{k_n} f_{sharing}(\vec{k}^2, B_i^n) \\ &= (k_n - 1) \prod_{j=1}^{n-1} L_j + \sum_{l=1}^{n-1} ((k_l - 1) \times \prod_{j=1 \wedge j \neq l}^{n-1} L_j) \times \sum_{i=1}^{k_n} L^i \end{aligned}$$

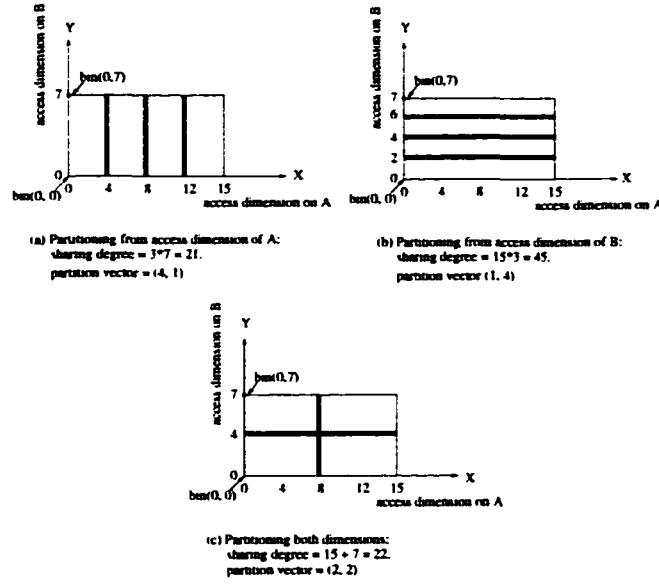


Figure 5.2: Partitioning patterns of an 2-dimensional bin space on four processors.

$$\begin{aligned}
 &= (k_n - 1) \prod_{j=1}^{n-1} L_j + \sum_{l=1}^{n-1} ((k_l - 1) \times \prod_{j=1 \wedge j \neq l}^n L_j) \\
 &= \sum_{l=1}^n ((k_l - 1) \times \prod_{j=1 \wedge j \neq l}^n L_j) \quad (5.9)
 \end{aligned}$$

This shows that the theorem is valid.

□

In Figure 5.2, the 2-dimensional bin space obtained in Figure 5.1 is partitioned into 4 partitions in three different ways where a partitioning vector is given in order (X, Y). With respect to the data-sharing degree, the partitioning method given in Figure 5.2(a) is the best.

5.3.2 An Algorithm to Determine Partitioning Vector

Based on the definition of a partitioning vector given in Definition 6, each element of a partitioning vector must be 1 or a factor of p , the number of processors. Assume that p

be factored into r prime factors. On an n -dimensional bin space, a partitioning vector has n elements. If we consider each prime factor of p as a ball and each element of a partitioning vector as a slot, then searching an optimal partitioning vector can be viewed as finding an optimal method to throw r balls into the n slots of a partitioning vector so that the resulted vector gives the smallest sharing degree. This is a classic combinational problem with NP-complete complexity [63]. Only an efficient approximate algorithm is practical.

The design of approximate algorithms for NP-complete problems has been widely studied in the area of Artificial Intelligence [58]. Exploiting heuristic information has been shown to be helpful for designing a good approximate algorithm. Motivated by the well-known “Hill Climbing” algorithm in artificial intelligence [58], we exploit some heuristic information to reduce the sharing degree while balls are thrown one-by-one into slots of a partitioning vector. This approach makes its best effort to approximate a better partitioning vector.

The idea behind our heuristic algorithm is to use an incremental approach to find a better partitioning vector. Based on a pre-determined partitioning vector \vec{k} which divides a bin space into d partitions, the algorithm tries to put a prime number q into \vec{k} to form a new partitioning vector which divides the bin space into $d \times q$ partitions with a smaller sharing degree. The procedure is repeated until all the prime factors of p , the number of processors, are put into the partitioning vector. In the following, we first present several properties before describing our algorithm.

Because the relative order among dimensions in a bin space does not affect the selection of a partitioning vector, we assume that the dimensions of a targeted bin space $B^n(0 : L_1, 0 : L_2, \dots, 0 : L_{n-1}, 0 : L_n)$ have been sorted in decreasing order from left to right, i.e., $L_i \geq L_{i+1}$ for $i = 1, 2, \dots, n - 1$.

Theorem 3 Ordering Rule

For a given partitioning vector $\vec{k}(k_1, k_2, \dots, k_n)$ not in decreasing order, the partitioning vector resulting by sorting \vec{k} in decreasing order is at least as good as \vec{k} in terms of the sharing degree.

Proof: Let $\vec{k}(k_1, k_2, \dots, k_r, k_{r+1}, \dots, k_n)$ be a partitioning vector which is not in decreasing order. If $k_r < k_{r+1}$ for an $r \in [1, n]$, we construct another partitioning vector $\vec{k}'(k_1, k_2, \dots, k_{r+1}, k_r, \dots, k_n)$ by swapping k_r with k_{r+1} . Based on Theorem 2, the sharing degrees of the partitions generated by the two partitioning vectors are:

$$\begin{aligned} f_{sharing}(\vec{k}, B^n) &= \sum_{i=1 \wedge i \neq r \wedge i \neq r+1}^n ((k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j) + (k_r - 1) \prod_{j=1 \wedge j \neq r}^n L_j \\ &\quad + (k_{r+1} - 1) \prod_{j=1 \wedge j \neq r+1}^n L_j. \end{aligned} \quad (5.10)$$

$$\begin{aligned} f_{sharing}(\vec{k}', B^n) &= \sum_{i=1 \wedge i \neq r \wedge i \neq r+1}^n ((k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j) + (k_r - 1) \prod_{j=1 \wedge j \neq r+1}^n L_j \\ &\quad + (k_{r+1} - 1) \prod_{j=1 \wedge j \neq r}^n L_j. \end{aligned} \quad (5.11)$$

Based on equations (5.10) and (5.11), we have

$$\begin{aligned} f_{sharing}(\vec{k}, B^n) &- f_{sharing}(\vec{k}', B^n) \\ &= (k_r - 1) \left(\prod_{j=1 \wedge j \neq r}^n L_j - \prod_{j=1 \wedge j \neq r+1}^n L_j \right) \\ &\quad + (k_{r+1} - 1) \left(\prod_{j=1 \wedge j \neq r+1}^n L_j - \prod_{j=1 \wedge j \neq r}^n L_j \right) \\ &= (k_r - 1)(L_{r+1} - L_r) \prod_{j=1 \wedge j \neq r \wedge j \neq r+1}^n L_j \\ &\quad + (k_{r+1} - 1)(L_r - L_{r+1}) \prod_{j=1 \wedge j \neq r \wedge j \neq r+1}^n L_j \\ &= (L_r - L_{r+1})(k_{r+1} - k_r) \prod_{j=1 \wedge j \neq r \wedge j \neq r+1}^n L_j \\ &\geq 0. \end{aligned} \quad (5.12)$$

This shows that the sharing degree of \vec{k}' equals to or is smaller than that of partitioning vector \vec{k} . Without increasing the sharing degree, this procedure can be repeated until the resulted vector is in decreasing order. So, the theorem is valid. \square

Based on the above ordering rule, we only focus on looking for those partitioning vectors whose elements are arranged in decreasing order. This greatly narrows the search range and reduces the overhead of the algorithm.

Theorem 4 Increment Rule 1

For an n -dimensional bin space B^n , and partitioning vectors $\vec{k}(k_1, k_2, \dots, k_i, k_{i+1} \times q, 1, \dots, 1)$ and $\vec{k}'(k_1, k_2, \dots, k_i \times q, k_{i+1}, 1, 1, \dots, 1)$, where $q > 1$, \vec{k} is better than \vec{k}' in terms of the sharing degree if and only if

$$k_i \times L_{i+1} > k_{i+1} \times L_i.$$

Proof: Based on Theorem 2, we have

$$\begin{aligned} f_{sharing}(\vec{k}, B^n) &= \sum_{l=1}^{i-1} ((k_l - 1) \prod_{j=1 \wedge j \neq l}^n L_j) + (k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j \\ &\quad + (q \times k_{i+1} - 1) \prod_{j=1 \wedge j \neq i+1}^n L_j. \\ f_{sharing}(\vec{k}', B^n) &= \sum_{l=1}^{i-1} ((k_l - 1) \prod_{j=1 \wedge j \neq l}^n L_j) + (q \times k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j \\ &\quad + (k_{i+1} - 1) \prod_{j=1 \wedge j \neq i+1}^n L_j. \end{aligned}$$

Based on the above two equations, we have

$$\begin{aligned} f_{sharing}(\vec{k}, B^n) - f_{sharing}(\vec{k}', B^n) &= (k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j + (q \times k_{i+1} - 1) \prod_{j=1 \wedge j \neq i+1}^n L_j \\ &\quad - (q \times k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j - (k_{i+1} - 1) \prod_{j=1 \wedge j \neq i+1}^n L_j \end{aligned}$$

$$\begin{aligned}
& -(q \times k_i - 1) \prod_{j=1 \wedge j \neq i}^n L_j - (k_{i+1} - 1) \prod_{j=1 \wedge j \neq i+1}^n L_j \\
= & ((k_i - 1)L_{i+1} + (q \times k_{i+1} - 1)L_i - (k_i \times q - 1)L_{i+1} \\
& - (k_{i+1} - 1)L_i) \prod_{j=1 \wedge j \neq i \wedge j \neq i+1}^n L_j \\
= & (q - 1)(k_{i+1}L_i - k_iL_{i+1}) \prod_{j=1 \wedge j \neq i \wedge j \neq i+1}^n L_j. \tag{5.13}
\end{aligned}$$

Based on $q > 1$ and equation (5.13), $f_{sharing}(\vec{k}, B^n)$ is smaller than $f_{sharing}(\vec{k}', B^n)$ if and only if $k_{i+1}L_i - k_iL_{i+1} < 0$, i.e., $k_iL_{i+1} > k_{i+1}L_i$. So, the theorem is valid. \square

For a given bin space which has been partitioned by $\vec{k}(k_1, k_2, \dots, k_i, k_{i+1}, 1, \dots, 1)$, if the i -th dimension or the $(i+1)$ -th dimension is further partitioned by q , the increment rule tells which dimension should be chosen. Based on the above theorem, the following corollary can be directly obtained.

Corollary 3 Increment Rule 2

For an n -dimensional bin space B^n , and partitioning vectors $\vec{k}(k_1, k_2, \dots, k_i, k_{i+1}, 1, \dots, 1)$ and $\vec{k}'(k_1, k_2, \dots, k_i \times k_{i+1}, 1, 1, \dots, 1)$, where $k_{i+1} > 1$, \vec{k} is better than \vec{k}' in terms of the sharing degree if and only if

$$k_i \times L_{i+1} > k_{i+1}L_i.$$

Based on the above three rules, we design an efficient heuristic algorithm as follows.

1. Factor p , the number of processors, to generate all the prime factors of p in decreasing order. Assume that there are q prime factors: $r_1 \geq r_2 \geq \dots \geq r_q$. Initially, the n -dimensional partitioning vector \vec{k} , stored in $k[1 : n]$, is $(1, 1, \dots, 1)$ for the bin space $B^n(0 : L_1, 0 : L_2, \dots, 0 : L_n)$.

2. Let $last$ index the position in $k[1 : n]$ where $k[i] > 1$ for $i < last$ and $k[i] = 1$ for $i \geq last$. Initially, $last = 1$. For each prime factor r_j where j increases from 1 to q , do the following:
 - (a) When $(last \leq n)$, use the increment rule 2 to determine whether r_j should be put in $k[last]$. Based on the ordering rule, the best place to put r_j must be in $k[1 : last]$. So, we use increment rules to find a better place in $k[1 : last]$. If so, $last$ is increased by 1 and go back; otherwise, use the increment rule 1 to put r_j together with $k[last - 1]$ or $k[last - 2]$, then reorder $k[1 : last - 1]$ in decreasing order and go back.
 - (b) Otherwise: use the increment rule 1 to put r_j together with $k[last - 1]$ or $k[last - 2]$, then reorder $k[1 : last - 1]$ in decreasing order and go back.

The factorization procedure can be finished in $O(\sqrt{p})$. Putting prime factors in decreasing order helps to reduce the overhead of sorting the partitioning vector. In the best case where the above algorithm does not conduct sorting, the algorithm has computational complexity $O(n + \sqrt{p})$. In the worst case, the algorithm finishes in $O(n^2 + \sqrt{p})$. In practical cases, n , the number of arrays, usually is not larger than 4. In addition, the number of processors in a SMP system is often smaller than 16. So, this partitioning vector determination algorithm is efficient. Because the above heuristic algorithm does not guarantee an optimal solution, it can be further improved by using additional heuristic information at the cost of higher overhead. However, whether a more precise algorithm would further improve performance is still an open question.

5.3.3 Bin Space Partitioning Procedure

Let (k_1, k_2, \dots, k_n) be the partitioning vector determined by the previous section. Here, we explain how to use the partitioning vector to partition the n -dimensional bin space,

$(0 : \lceil \frac{s_1-1}{fC/n} \rceil, 0 : \lceil \frac{s_2-1}{fC/n} \rceil, \dots, 0 : \lceil \frac{s_n-1}{fC/n} \rceil)$, that is generated in Section 5.2. Based on the partitioning, a set of transformation functions and the internal hash structure are obtained.

The i -th dimension of the n -dimensional bin region $(0 : \lceil \frac{s_1-1}{fC/n} \rceil, 0 : \lceil \frac{s_2-1}{fC/n} \rceil, \dots, 0 : \lceil \frac{s_n-1}{fC/n} \rceil)$ may not be evenly divided by k_i , for $1 \leq i \leq n$. Let r_i be the reminder ($i = 1, 2, \dots, n$), which can be calculated as:

$$r_i = (\lceil \frac{s_i-1}{fC/n} \rceil + 1) \bmod k_i \text{ for } i = 1, 2, \dots, n,$$

where \bmod is the modulus operation.

We calculate L_i as follows:

$$L_i = \lceil \frac{\lceil \frac{s_i-1}{fC/n} \rceil + 1}{k_i} \rceil, \text{ for } i = 1, 2, \dots, n$$

Using the even partitioning method, the i -th dimension will be divided into k_i parts, where r_i of them have length L_i and the others have length $L_i - 1$. The concrete partitioning method is shown in the following, where the i -th dimension, $[0, \lceil \frac{s_i-1}{fC/n} \rceil]$, of the n -dimensional bin region is partitioned into k_i mostly even parts (for $i = 1, 2, \dots, n$ respectively):

$$\underbrace{[0, L_i - 1] \cdots [(r_i - 1)L_i, r_i L_i - 1]}_{r_i \text{ parts of length of } L_i} \underbrace{[r_i L_i, (r_i + 1)L_i - 2] \cdots [(k_i - 1)L_i - k_i + r_i + 1, k_i L_i - k_i + r_i - 1]}_{(k_i - r_i) \text{ parts of length of } L_i - 1}$$

In the above, all the parts are relabeled from 0 to $k_i - 1$ (The correctness of the above expression can be verified by $(k_i L_i - k_i + r_i - 1) = \lceil \frac{s_i-1}{fC/n} \rceil$). Based on the numbers of the parts, each of the $\prod_{j=1}^n k_j$ ($= p$) partitions generated can be expressed as an index vector (i_1, i_2, \dots, i_n) , where i_j ($1 \leq j \leq n \wedge 0 \leq i_j \leq k_i - 1$) is its part label in the i -th dimension.

In Figure 5.3(a), the bin space generated in Section 5.2 is partitioned by the partitioning vector $(2, 2)$. The generated 4 partitions are indexed as $(0,0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$.

The mapping function from a point (x_1, x_2, \dots, x_n) in the bin space to the index of its partition is

$$f_3(x_1, x_2, \dots, x_n) = (h1(x_1, 1), h1(x_2, 2), \dots, h1(x_n, n)), \quad (5.14)$$

where the function $h1$ is defined as follows:

$$h1(x, i) = \begin{cases} \lfloor \frac{x}{L_i} \rfloor & \text{if } x < r_i L_i \text{ or } r_i = 0 \\ r_i + \lfloor \frac{x - r_i L_i}{L_i - 1} \rfloor & \text{otherwise} \end{cases} \quad (5.15)$$

To reorganize the partitions, each partition is constructed as an independent n -dimensional space using its smallest point (that has smallest coordinate in each dimension) as its origin point $(0, 0, \dots, 0)$. For each point (x_1, x_2, \dots, x_n) in a partition, its new coordinate in the independent space of the partition is given by

$$f_4(x_1, x_2, \dots, x_n) = (h2(x_1, 1), h2(x_2, 2), \dots, h2(x_n, n)), \quad (5.16)$$

where function $h2$ is

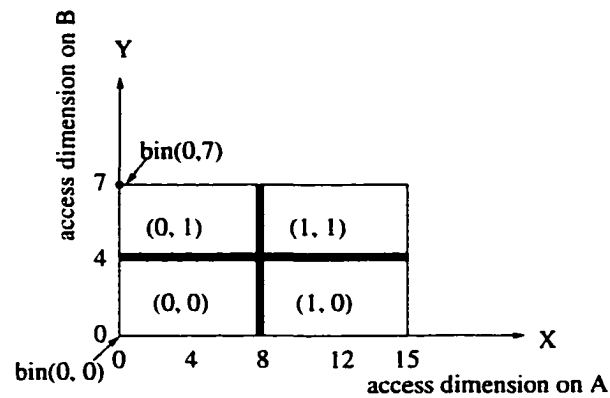
$$h2(x, i) = \begin{cases} x \bmod L_i & \text{if } x < r_i L_i \text{ or } r_i = 0 \\ (x - r_i L_i) \bmod (L_i - 1) & \text{otherwise} \end{cases} \quad (5.17)$$

In Figure 5.3(b), each partition is transformed as an independent space by function f_4 .

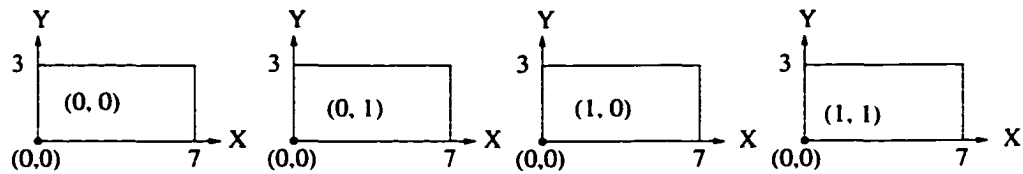
Based on the indices of the partitions, the p partitions are further reorganized in a new dimension, the $(n + 1)$ -th dimension, by the following hash function, which maps a partition with index (i_1, i_2, \dots, i_n) into a coordinate on the $(n + 1)$ -th dimension $[0, \prod_{j=1}^n k_j - 1]$:

$$f_5(i_1, i_2, \dots, i_n) = \sum_{j=1}^{n-1} i_j k_j + i_n. \quad (5.18)$$

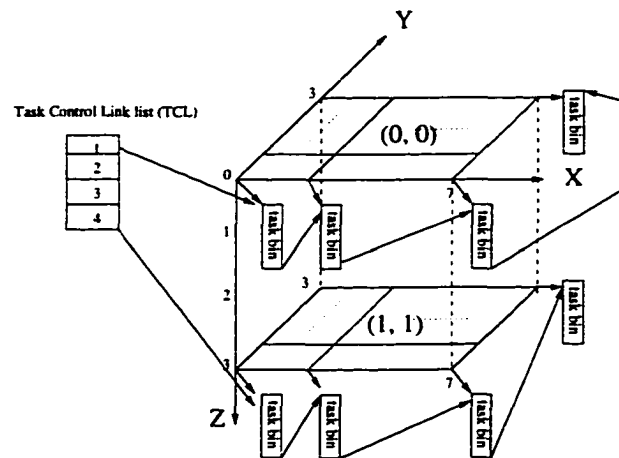
This will create an $(n + 1)$ -dimensional space. The coordinate (x_1, x_2, \dots, x_n) of a point in the original n -dimensional independent space of a partition with index (i_1, i_2, \dots, i_n) is extended by function F as follows,



(a) Indexing of partitions.



(b) independent address space of each partition.



(c) 3-dimensional internal representation of the memory access space.

Figure 5.3: Indexing partitions: the bin space is evenly divided into 4 partitions from X and Y dimensions.

$$F(x_1, x_2, \dots, x_n) = (x_1, x_2, \dots, x_n, f_5(i_1, i_2, \dots, i_n)). \quad (5.19)$$

Based on the created $(n + 1)$ -dimensional structure, we can build an $(n + 1)$ -dimensional hash structure, which is $(0 : L_1 - 1, 0 : L_2 - 1, \dots, 0 : L_n - 1, 0 : p - 1)$. For a given task $t_i(a_{i1}, a_{i2}, \dots, a_{in})$ with n hints, the position onto which the task is mapped is given as follows by hash functions f_1, f_2, f_3, f_4, f_5 , and F that are defined by equations (5.1), (5.2), (5.14), (5.16), (5.18), and (5.19) respectively:

$$F(f_4(f_2(f_1(a_{i1}, a_{i2}, \dots, a_{in}))),$$

where function F will call functions f_3 and f_5 . The above mapping has complexity of $O(n)$, which is independent of the total number of tasks. Because n , the number of hints, is a small constant, the mapping approximately finishes in a constant time. Although the whole transformation procedure seems to be complicated, the derived hash structure and hash functions provide a very efficient implementation. When all the tasks are created at run-time, they are automatically mapped into the $(n + 1)$ -dimensional hash structure to form p partitions: $(0 : L_1 - 1, 0 : L_2 - 1, \dots, 0 : L_n - 1, 0)$, $(0 : L_1 - 1, 0 : L_2 - 1, \dots, 0 : L_n - 1, 1)$, \dots , $(0 : L_1 - 1, 0 : L_2 - 1, \dots, 0 : L_n - 1, p - 1)$.

5.4 Putting Them Together in An Implementation

To assist the dynamic scheduling of created tasks, all tasks are organized in groups with a fixed size. All task groups in a bin are chained together. Then all the group chains in bins are linked together in the creation order of bins. An auxiliary data structure, called a Task Control Linked list (TCL list), is constructed. The TCL list is a structure array with p elements corresponding to the p partitions, respectively, in the hash structure. Each element of the TCL list has a task counter which gives the number of tasks in the corresponding partition, and two pointers, *head* and *tail*, respectively, pointing to the

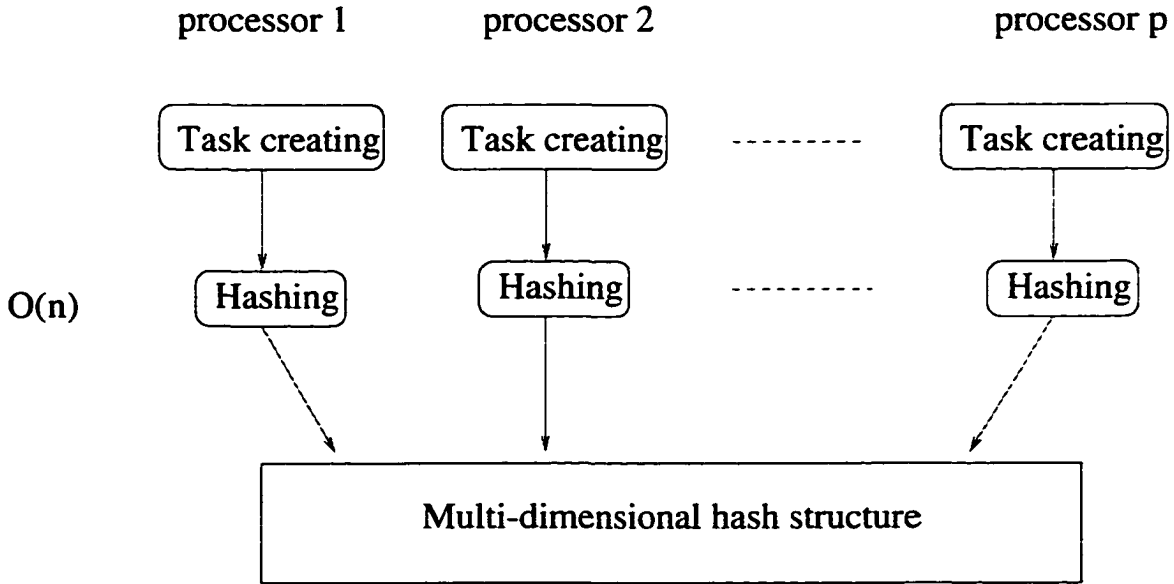


Figure 5.4: Flowchart of task reorganization in the run-time system.

head and the tail of the task group chain in the corresponding partition. For example, in Figure 5.3(c), a 3-dimensional hash structure is constructed from the partitions produced in the partitioning phase. All the task bins in a partition are chained together based on their creation order at run-time.

The multi-dimensional hash structure and the related hash functions are built based on the hints provided by initialization function `cacheminer_init`. Tasks are created in parallel by multiprocessors and are mapped into an appropriate task bin in an appropriate partition by hash functions. The locality optimizations are implicitly conducted in the task mapping procedure with $O(n)$ complexity. Because n , the number of arrays, is usually smaller than 4, which is independent of the number of tasks, the reorganization procedure of N tasks can be considered to finish in $O(N)$. The working flowchart of the run-time system is illustrated in Figure 5.4.

Chapter 6

Task Scheduling at Run-time

The dynamic scheduler in the run-time system is aimed at minimizing the parallel computing time of a set of data-independent tasks created in the locality optimization phase described in the previous chapter. The partitions generated in the task partitioning step are balanced in the sense that all the partitions access almost the same size of memory address space for each array, which is locality oriented. However, this may not guarantee that all the partitions have the same execution time, due to the possible effects of the following factors: (1) irregular data-access patterns in the tasks, which will result in imbalanced numbers of tasks allocated among the partitions; (2) irregular computation patterns in the tasks, which will directly result in different execution times for the partitions; (3) the interference of a paging system in the operating system, which also results in imbalanced numbers of tasks allocated among the partitions; (4) the quantity difference of data locality exploited among different partitions, which may execute different partitions at different rates.

The run-time scheduling of parallel iterations or tasks (here, “iterations” refers to tasks in our abstraction given in Chapter 5) in shared-memory systems has been intensively investigated for ten years. This chapter first analyzes the design principles and

limitations of existing work, which motivate the design of more efficient scheduling algorithms. Then, an adaptive scheduling algorithm and its several variations are presented. The proposed algorithm and its variations are experimentally evaluated with respect to previous algorithms. Finally, a locality-preserved scheduling algorithm is presented to schedule the tasks in the partitions generated from locality-oriented optimizations in previous chapter.

6.1 Overview of Existing Work and Motivation

The performance of a task scheduling algorithm is mainly affected by three overhead sources: *synchronization and loop allocation*, *load imbalance*, and *data communication*. Although it is desirable that an efficient algorithm minimizes the above three sources of overhead, this is usually impossible because conflicts can arise among them. Exploiting processor affinity (processor affinity refers to certain data access dependency of a task to a specific processor, a more precise definition is given in Section 6.4.1) favors the allocation of loop iterations close to their data, which tends to cause load imbalance. Load balance favors the “fine grain” allocation of loop iterations (where a small number of iterations are allocated) in order to minimize the effects of uneven assignment. However, the “fine grain” allocation tends to increase synchronization overhead and loop allocation overhead. In different applications, each overhead source affects performance differently. Hence, an efficient loop scheduling algorithm should optimize its performance by adaptively trading off synchronization overhead, loop allocation overhead, load imbalance overhead and data-communication overhead. Moreover, a dynamic scheduling algorithm should not assume any prior knowledge of the execution times of the loop iterations because the execution of the loop usually is unpredictable in practice.

So far, many novel dynamic scheduling algorithms have been proposed , e.g. [32, 49, 47, 51, 61, 68, 77, 69]. These algorithms fall into two distinct classes: central

work queue based and distributed work queue based. In the central work queue-based algorithms [32, 61, 77, 69], iterations of a parallel loop are all stored in a shared central work queue and each processor exclusively grabs some iterations from the central queue to execute. The major advantage of using a central work queue is the possibility of evenly balancing the workload. While keeping a good load balance, the central work queue based algorithms differ in the way they reduce synchronization and loop allocation overheads. However, three limitations are associated with the use of a central work queue: (1) An iteration in the central work-queue is likely to be dynamically allocated to execute on any processor, which does not facilitate the exploitation of processor affinity; (2) During allocation, all the processors but one should remotely access the central work queue, and thereby generating heavy network traffic; (3) Because all the processors contend for the central work queue, the central work queue tends to be a performance bottleneck, resulting in a longer synchronization delay.

In order to exploit the processor affinity inherent in the parallel execution of many loops and to eliminate the central bottleneck, the affinity scheduling algorithm proposed in [51] distributes the central work queue to be local to each processor, and the algorithm partitions iterations of a parallel loop statically into local work queues so that each processor is only involved in remote access when load imbalance occurs. Markatos and Leblanc [51] show that affinity scheduling almost always achieves the best performance in all tested cases when compared with central work queue based algorithms. To enhance the affinity scheduling algorithm in the presence of a large, correlated imbalance in loop execution time, Subramaniam and Eager [68] propose two loop partition methods: dynamic partition and wrapped partition. These two partition methods, however, only improve affinity scheduling for some specific applications because both of them execute under some specific assumptions about the distribution of loop execution time.

In the design of distributed work queue based algorithms, we have no reason to prefer other kinds of loop partition methods to a uniform partition, due to the uneven

and unpredictable execution time of loop iterations. Hence, it is crucial for a distributed work queue based algorithm to be able to schedule tasks dynamically and efficiently at run-time to even the load imbalance caused by a static partition. In existing affinity scheduling algorithms [51, 68], all the processors schedule loop iterations in their local queues using the same allocation scheme where, at each time, $1/P$ of the remaining iterations in the local queue are allocated (P is the number of processors). This iteration allocation scheme may not be efficient. For example, if the initial loop partition is balanced, then all processors will complete the execution of iterations in their local queues at the same time, and each processor should grab all the iterations in the local queue in an allocation, instead of only $1/P$ of the remaining iterations. On the other hand, if the initial loop partition is not balanced, those lightly loaded processors should finish execution of the iterations in their local queue as soon as possible so that they can immediately turn to help heavily loaded processors. Hence, processors should be able to dynamically increase or decrease their allocation granularity based on runtime information to reduce synchronization and loop allocation overhead and balance load more evenly. This motivates us to design adaptive scheduling algorithms to further improve existing affinity scheduling algorithms.

Our main idea is to exploit the potential of dynamic information to reduce loop execution time. Regarding the general scheduling problem where the affinities among the data accesses of tasks have not been exploited, an adaptive scheduling algorithm and its five variations are proposed in this chapter. These algorithms dynamically adjust allocation granularity according to a program's execution history. Based on a set of fairly selected applications, the effectiveness of the proposed algorithms are compared with the existing affinity scheduling algorithms in [51, 68]. Based on this, a locality-preserved scheduling algorithm is designed.

6.2 An Adaptive Scheduling Algorithm

Similar to the affinity scheduling algorithm [51], the adaptive affinity scheduling algorithm is also constructed to have following three phases:

Initial partition phase: A deterministic assignment policy is used to partition iterations of a parallel loop into local queues of processors, which ensures that an iteration is always assigned to the same processor at the start. With this assignment scheme, if a parallel loop executes repeatedly, and each parallel iteration accesses the same data set in different executions, the first execution of the parallel loop will bring data locally to processors so that the subsequent execution of the parallel loop only involves local data access.

Local scheduling phase: Based on a local scheduling policy, each processor allocates a part of the remaining iterations in a local queue to execute until the local queue is empty. Local scheduling does not cause remote access overhead. Because each local queue is shared by all processors, a critical section is used to protect the allocation of the loop iterations in the local queue. The local scheduling overheads mainly come from the synchronization overhead and the loop allocation overhead in the execution of the critical section. Reducing the number of allocations is crucial to improve the performance of the local scheduling phase.

Remote scheduling phase: When a processor finishes the execution of all the iterations in the local queue, it remotely allocates for execution a portion of the iterations from the most loaded processor in the system. The remote scheduling phase is aimed at dynamically balancing the workload. An iteration is at most reassigned once, which avoids processor thrashing. Remote scheduling causes remote data access overhead as well as synchronization overhead and loop allocation overhead.

Instead of relying on pre-computed knowledge about a loop's execution, our adaptive affinity scheduling algorithm exploits the potential of using dynamic execution history to adaptively adjust iteration chunk size to reduce synchronization and loop allocation overheads. The algorithms also maintain a better load balance. The main idea of our designs is to minimize local scheduling overhead so that the phase of dynamically balancing the workload can be speeded up, which reduces loop execution time.

In the initial phase, a loop with N iterations is partitioned into chunks of uniform size $\lceil N/P \rceil$ over P processors, because we have no reason to prefer other partition methods in the absence of a precise prediction about the execution distribution of the loop's iterations. Our initial partition is identical to the initial one in [51].

In the local scheduling phase, a processing speed variable s_i , termed the *PS* variable, is set for each processor, which keeps track of the number of iterations the processor has executed so far ($i = 1, \dots, P$). The variable s_i is initially set to 0 and is increased by 1 each time processor i finishes the execution of an iteration. By comparing the local *PS* variable with other *PS* variables, a processor can observe its load distribution. At any time, the processors with smaller *PS* variable values have executed iterations that have a heavier workload than those executed by the processors with larger *PS* variable values. Because, in most applications, a load distribution state has a certain steady duration, it is feasible to speculate about the load distribution in the near future by the current observation. Although some applications really experience sudden changes in the load distribution, the prediction difference can be minimized by dynamic readjustments. In order that processors respond spontaneously to the dynamic changes of iteration workload, it is necessary to differentiate the workload states of processors. For fairness, we select the average number of iterations executed by all processors, i.e. $\sum_{i=1}^P s_i / P$, as the pivot to partition the workload states of processors into the following three types:

- heavily loaded (HL)— the processor's *PS* value is smaller than $\sum_{i=1}^P s_i / P - \alpha$;

- normally loaded (NL)— the processor's PS value is within the range of $(\sum_{i=1}^P s_i/P - \alpha, \sum_{i=1}^P s_i/P + \alpha)$; and
- lightly loaded (LL)— the processor's PS value is equal to or larger than $\sum_{i=1}^P s_i/P + \alpha$;

where α is a non-negative, range control variable, which adjusts the distribution of HL processors, NL processors and LL processors. Variations of parameter α would affect the algorithmic performance. The dynamic features in the execution of real applications' loops make it impractical or impossible to analytically determine an optimal value of α . Here, we will discuss the effect of different values of α on performance through experiments, giving an empirical method for determining a performance-efficient α .

In order to control the chunk size in the allocation of loop iterations, a chunk-size control variable k_i is set for processor i ($i = 1, \dots, P$). Each processor always removes $1/k_i$ of the remaining iterations in its local work queue for execution. In the beginning of the local scheduling phase, all of the chunk-size control variables are initialized to the same value, such as P , the total number of processors. Then, each k_i ($i = 1, \dots, P$) is adaptively and independently adjusted by a chunk-size control function Π . The function Π uses the load state and the current value of k_i of processor i as its two input parameters, and adjusts the chunk-size control variable k_i as follows:

- If processor i is heavily loaded, Π increases k_i , aiming at reducing chunk size so that more iterations remaining in the heavily loaded processor can be executed by those lightly loaded processors, therefore balancing workload more efficiently.
- If processor i is normally loaded or lightly loaded, Π decreases k_i , aiming at increasing chunk size so that a normally loaded or lightly loaded processor can finish all the iterations in its local work queues as soon as possible, and then immediately start to help heavily loaded processors.

When processor i completes execution of the iterations in the local queue, it turns to the remote scheduling phase. In the affinity scheduling algorithm of [51], when a processor exhausts its local work queue and starts to help other heavily loaded processors, it just removes $\lceil 1/P \rceil$ of the remaining iterations from the most heavily loaded processor. This allocation method may not be efficient when only a few processors can turn to help other processors. Here, we determine the chunk size according to the current number of lightly loaded or normally loaded processors, because they are able to help those heavily loaded processors in the near future. Processor i determines its chunk control variable k_i as follows:

$$k_i = \min\{P, n + 1\}, \quad (6.1)$$

where n is the total number of lightly loaded processors and normally loaded processors ($n + 1$ means to include the most heavily loaded processor from which processor i will allocate the remaining iterations), and P is the total number of processors. Then, processor i allocates $\lceil 1/k_i \rceil$ of the remaining iterations in the most heavily loaded processor to execute. This procedure will repeat until all local work queues are empty. Initially, k_i has a smaller value than P so that a big chunk size is used to reduce the number of remote allocations. Our experiments in the next section will show that the selected big chunk size does not increase the risk of imbalancing load. Subsequently, when more processors become lightly loaded or normally loaded, k_i will increase until it reaches the maximal value, P .

In the following, a pseudocode description of the adaptive affinity scheduling algorithm is given. In implementation, this code can be automatically inserted by a compiler into application programs for each processor to dynamically schedule the execution of loops without the interference of the operating system. As with other existing work, we generate dynamic scheduling programs by hand in our experiments in order to focus on the algorithmic study.

1. Initial partition phase:

initial_partition(N, P) // N iterations are uniformly partitioned over P processors

```
{for ( $i = 0; i < P; i++$ )
    assign_iterations( $i$ ); // assign iterations to processor  $i$ .
for ( $i = 0; i < P; i++$ ){
     $s_i = 0; k_i = P$ ; // Initialize PS variables and  $K'_i$ 's
}
```

2. Locally scheduling phase:

```
loop { // processor  $i$  gets  $1/k_i$  of the local iterations to execute and adjusts  $k_i$ .
    Lock(local_queue. $i$ );
    range = get_iterations(local_queue. $i$ ,  $1/k_i$ ); // allocate  $1/k_i$  of the iterations.
    unlock(local_queue. $i$ );
    While (range -- != 0) {
        execute_one_iteration();  $s_i++$ ;
    }
    state = load_state(( $\sum_{i=1}^P s_i$ )/ $P, s_i, \alpha$ ); //compute the load state of processor  $i$ .
     $k_i = \Pi(state, k_i)$ ; //adjust the chunk granularity
} until (local_queue. $i = \emptyset$ )
```

3. Remotely scheduling phase:

```
 $k_i = 1$ ;
loop { if ( $k_i \neq P$ ) {
     $k_i = \text{find\_non\_heavily\_loaded\_processor}()$ ;  $k_i = \min\{P, k_i + 1\}$ ;
     $j = \text{find\_most\_loaded\_processor}()$ ;
    lock(local_queue. $j$ );
    range = get_iterations( $j, 1/k_i$ ); //get  $\lceil 1/k_i \rceil$  of the iterations on processor  $j$ .
    unlock(local_queue. $j$ );
    execute(range);
}
```

} until (all iterations have been finished).

Adaptively changing loop scheduling granularity is the major characteristic which distinguishes our adaptive affinity scheduling from the affinity scheduling algorithm in [51]. Remotely reading the *PS* variables of other processors is the overhead caused by our adaptive scheduling algorithm in collecting execution history of other processors. If the increased overhead nullifies the benefit of adaptively varying loop scheduling granularity, the adaptive affinity scheduling algorithm may not exhibit a performance improvement over existing affinity scheduling algorithms.

6.3 Variations of the Adaptive Scheduling Algorithm

Different variations of the adaptive affinity scheduling algorithm can be constructed by designing different chunk-size control protocols for the function Π . Here we propose four mechanisms for our adaptive algorithm. Let k_i be the chunk size control variable of processor i in the local scheduling phase.

- **Exponential Adaptive (EA) Mechanism**

In the EA mechanism, a processor increases or decreases the value of its chunk-size control variable k_i by a factor at each time according to the current load state. The chunk-size control function Π is formally defined as

$$\Pi(\text{state}, k_i) = \begin{cases} k_i * \text{base} & \text{if state=HL} \\ \lceil k_i / \text{base} \rceil & \text{if state=NL or LL} \end{cases}$$

where *base* is an integer constant. Here, we choose 2 for *base*. Initially, k_i is set to P in our adaptive algorithm.

- **Linear Adaptive (LA) Mechanism**

In the LA mechanism, a processor increases or decreases its chunk size control

variable k_i by a constant at each time interval according to the current load state. The chunk-size control function Π is formally defined as

$$\Pi(\text{state}, k_i) = \begin{cases} k_i + \text{con} & \text{if state=HL} \\ \max\{1, k_i - \text{con}\} & \text{if state=NL or LL} \end{cases}$$

where con is a constant specified by the user. We choose 1 in our experiments. Because the LA mechanism changes the chunk size at a slower pace than the EA algorithm, it has less risk of imbalancing the workload, but larger synchronization and loop allocation overhead.

- **Conservation Adaptive (CA) Mechanism**

A careful selection of the chunk size in a loop scheduling algorithm is crucial to find a compromise between synchronization overhead and load imbalance. Allocating a bigger chunk of the iterations of a loop tends to reduce synchronization and loop allocation overhead, but increase the risk of imbalancing load. Previous work in [47] shows that in order to have reasonable load imbalance and synchronization overhead, it is safe to choose a value in $[P, 2P]$ for the chunk size control variable k_i . The CA mechanism is constructed by restricting the varying range of the chunk-size control variables of the LA mechanism within $[P/2, 2P]$. The chunk size control function is defined as follows:

$$\Pi(\text{state}, k_i) = \begin{cases} \min\{2P, k_i + \text{con}\} & \text{if state=HL} \\ \max\{P/2, k_i - \text{con}\} & \text{if state=NL or LL} \end{cases}$$

where con is a constant in $[0, P]$. We will use 1 in our experiments.

- **Greedy Adaptive (GA) Mechanism**

The GA mechanism employs a two-phase consensus method to greedily enlarge the chunk size on non-heavily loaded processors. The GA mechanism records the previous load state of the processor. If a processor finds it is in a non-heavily

loaded state in two consecutive allocations, it greedily reduces the chunk-size control variable to 1, i.e. it grabs all the remaining iterations in the local work queue to execute. Otherwise, the processor increases or decreases the chunk size by using the conservation method in the CA mechanism with $con = 1$ experimentally.

Let S_{pre}^i record the previous load state of processor i , and let S_c record the current load state of processor i . The chunk size control function is

$$\Pi(S_c, k_i) = \begin{cases} \min\{2P, k_i + con\} & \text{if } S_c = \text{HL} \\ \max\{P/2, k_i - con\} & \text{if } S_c \neq \text{HL and } S_{pre}^i = \text{HL} \\ 1 & \text{if } S_c \neq \text{HL and } S_{pre}^i \neq \text{HL} \end{cases}$$

Keeping and maintaining the PS variable for each processor allows the above four adaptive mechanisms to know exactly the current workload of each processor; thereby the PS variables can be used to adjust the speed for each processor, and as a consequence, to adjust the workload among the processors. But it also introduces loop allocation overhead.

Here we design a heuristic variation, denoted by HA, which still adopts the framework of our adaptive scheduling algorithm. Instead of using PS variables to determine workload distribution among processors, we use the number of iterations actually executed by each processor to guide the adjustments of scheduling granularities. Initially, a parallel loop is uniformly distributed to processors. Each processor i repeats grabbing $1/k_i$ of the remaining iterations in its local queue to execute without doing any adjustment to k_i . If processor i finishes all the iterations in its local work queue, and turns to get iterations from the most heavily loaded processor j , processor i is said to be lightly loaded and processor j is heavily loaded. Then processor i increases its scheduling granularity and processor j decreases its scheduling granularity. Hence, the lightly loaded processors can turn as early as possible to help heavily loaded processors, and the heavily loaded processors can release as much workload as possible to even those

lightly loaded processors. At the end of each execution of the parallel loop, processors check whether they have executed approximately the same number of iterations, i.e. a balanced workload. If so, processors increase their scheduling granularities to speed up subsequent executions of the parallel loop.

Comparing with the four variations of the adaptive algorithm: EA, LA, CA, and GA, the HA variation differs in several aspects: (1) Instead of determining the load state at each time of local scheduling that is used by the adaptive algorithms, the HA variation updates the load states of processors only in the remote scheduling phase and at the end of one execution of the parallel loop, so that it causes less scheduling overhead than the adaptive algorithm. (2) the HA variation works by requiring that the parallel loop be nested in a sequential loop to execute repeatedly. When the parallel loop only executes once, the HA variation becomes the affinity algorithm [51].

The pseudocode of the heuristic variation of the adaptive affinity scheduling algorithm (HA variation) is shown as follows.

1. Initial partition phase:

`initial_partition(N, P)` // N iterations are uniformly partitioned over P processors

{for ($i = 0; i < P; i++$)

`assign_iterations(i);` // assign iterations to processor i .

for ($i = 0; i < P; i++$)

$k_i = P;$

}

2. Locally scheduling phase:

loop { `Lock(local_queue. i);`

$range = \text{get_iterations}(i, 1/k_i);$ //allocate $1/k_i$ of the remaining iterations

`unlock(local_queue. i);`

`execute(range);`

```
} until (local_queue_i= $\emptyset$ )
```

3. Remotely scheduling phase:

```
loop {j = find_most_loaded_processor();
    lock(local_queue_j);
    range = get_iterations(j, 1/k_j); //get  $\lceil 1/k_j \rceil$  of the iterations of processor j.
    unlock(local_queue_j);
    if ( $k_i > 1$ )  $k_i = k_i - 1$ ; //increase the chunk size for processor i.
    if ( $k_j < 2 * P$ )  $k_j = k_j + 1$ ; //decrease the chunk size for processor j.
    execute(range); }
```

4. The program section at the end of each parallel loop:

```
end_para_loop
{
    barrier(&barrier, &P);
    if (tid == 0) //only processor 0 executes the code.
        find_maximum_and_minimum_of_chunk_sizes(kmax, kmin);
    if ((kmax - kmin) < P/2) //if the workload is balanced, increase chunk size.
        for (each processor i with  $k_i > 1$ )  $k_i = k_i/2$ ;
    barrier(&barrier, &P);
}
```

6.4 Evaluation Methods for Scheduling Algorithms

Markatos and Leblanc [51] show that the affinity scheduling algorithm (hereafter, simplified as the ML algorithm) outperforms other algorithms that do not exploit processor affinity. Hence, we focus on comparing the variations of the adaptive scheduling algorithm with the affinity scheduling algorithm and its two variations in [68]. The scheduling algorithms we have evaluated and compared are : (1) the ML affinity scheduling algorithm, (2) the SE dynamic initial partition affinity scheduling algorithm, (3) the adaptive

affinity algorithm with the exponential adaptive mechanism (EA), (4) the adaptive affinity algorithm with the linear adaptive affinity mechanism (LA), (5) the adaptive affinity algorithm with the conservative adaptive mechanism (CA), (6) the adaptive affinity algorithm with the greedy adaptive mechanism (GA), and (7) the heuristic adaptive variation (HA).

The experiments were conducted on one CC-NUMA machine and one SMP machine: the KSR-1, a hierarchical-ring-based, Cache Coherent Non-Uniform Memory Architecture (CC-NUMA), and one node of the HP SP2000 machine, a crossbar and ring-based cache coherent symmetric multiprocessor. The detailed description on the KSR-1 has been presented in our previous research work in [89]. The detailed information on a node machine of SP2000 is described in Chapter 7. Here, we address our methods of selecting application kernels and of evaluating the scheduling algorithms. Because the proposed adaptive algorithm and its variations are intended to be applied for a wide range of programming models, the performance evaluation of scheduling algorithms is not restricted by the programming model given in Chapter 3.1, which is locality-exploitation oriented.

6.4.1 Principles for Selecting Application Kernels

Considering the effects of program features on scheduling algorithms, we characterize parallel loops by three factors: the affinity of loop iterations to processors, the distribution of loop execution time, and the granularity of loop iterations.

Without exploiting the relations among the data accesses of parallel iterations, a scheduling algorithm can only exploit a kind of weak processor affinity that is exhibited by parallel iterations when they are going to be repeatedly executed. When a parallel iteration is executed many times, its multiple executions are expected to access the same set of data. This implies that an iteration should be allocated to execute on a fixed

processor so that memory accesses can be reduced. Hence, we first classify parallel loops into two classes: potential affinity parallel loops that are nested in a sequential loop, and non-affinity parallel loops that are only executed once. The strength of the affinity of the iterations as a potential affinity loop to a processor is significantly affected by the sizes of data sets accessed by iterations and by the data locality of iterations.

Based on our analysis of cache locality in section 3.3, we know that better data locality of an iteration means that the data set accessed by the iteration changes less significantly in different executions of the iterations. Data locality determines the affinity of iterations to processors. On the other hand, the sizes of data sets accessed by iterations determine the benefit of exploiting processor affinity. For those parallel loops with better data locality and iterations having very small data sets (e.g., one integer), exploiting processor affinity will not improve the execution times of these parallel loops more than by balancing load and reducing synchronization overhead.

As indicated in section 3.3, the cache locality is hard to be quantified precisely by a simple mathematic formula. Here, we use a simple model to measure approximately the locality of an iteration in multiple executions. Let $D(i)$ be the data set of an iteration in the i -th execution of a parallel loop, and let $|D(i)|$ be the size in bytes of data set $D(i)$. Then, $\bar{D} = \sum_{i=1}^N |D(i)|/N$ is the average size of data sets of the iteration over N executions of the parallel loop, and $\bar{\delta} = \sum_{i=1}^N |D(i) - D(i-1)|/N$, ($D(0)=D(1)$) is the average size difference of two data sets in two consecutive loop executions of an iteration. The value $\bar{\delta}$ indicates approximately how much data should be reloaded in each execution of an iteration of the parallel loop. So, the data locality of an iteration in multiple executions can be approximately quantified by the following defined locality rate:

$$\text{locality_rate} = 1 - \frac{\bar{\delta}}{\bar{D}},$$

where the locality rate is a value in $[0,1]$. A locality rate of 1 means an iteration always

accesses the same set of data. A larger locality rate represents a better data locality. Then, the strength of the affinity of an iteration to a processor can be quantitatively evaluated by $|\text{locality_rate} \times \bar{D}|$, the average number of data sets that will be accessed repeatedly (these data sets may be always stored in a local cache). In the selection of potential affinity parallel loops, we use data locality and data size to differentiate the affinity of iterations to processors.

The unpredictable variance in the execution times of parallel loops is a major obstacle for loop scheduling algorithms to work efficiently. In order to show how much parallel loop scheduling algorithms can tolerate different distributions of workload among iterations, we selected parallel loops by load distribution to cover three distinguished types of loops: (1) balanced loops where each iteration has the same amount of computation time, (2) predictable imbalanced loops where the computation times of iterations of a parallel loop vary as a predictable function of the loop control variable or where the load distribution in a parallel loop is fixed when it executes repeatedly, and (3) unpredictable imbalanced loops where the computation times of iterations change randomly, depending on initial input and some runtime variables (e.g., the execution time of a branch statement depends on its actual execution path). The ML algorithm only handles load imbalance by remote scheduling. The SE algorithm improves the ML algorithm performance only for those predictable imbalanced parallel loops where the load distribution of a parallel loop is not changed in multiple executions of the parallel loop, and the execution times of the iterations of a loop increase or decrease monotonically with the loop control variable. Our adaptive algorithm and its variations dynamically adjust the loop scheduling granularity to speed up the load balance procedure based on the execution history of processors. In the following experiments, we will show that the adaptive algorithm can handle load imbalance more efficiently over a wider range than the ML and the SE algorithms.

Besides affinity and load distribution, the iteration granularity of a loop is an-

other important factor affecting the performance of loop scheduling algorithms. For parallel loops with coarse granularity where the execution times of loop iterations are significantly larger than the overhead of remote access delay, balancing the workload is more crucial than reducing synchronization and loop allocation overheads. For parallel loops with fine granularity where the execution times of loop iterations are much smaller than the overhead of remote access delay, it is important to minimize scheduling overheads. Because the determination of the iteration granularity of a parallel loop depends on the interaction between the parallel loop and the underlying system, it is difficult to tell whether a parallel loop is coarsely grained or finely grained before execution. Instead of classifying parallel loops by granularity, we consider the effect of iteration granularity in our experiments.

Based on the above analyses, we classify parallel loops into six types by their affinity and load distributions: (I) loops with potential affinity and balanced workload, (II) loops with potential affinity and predictable workload, (III) loop with potential affinity and unpredictable workload, (IV) loops with non-affinity and balanced workload, (V) loops with non-affinity and predictable workload, and (VI) loops with non-affinity and unpredictable workload. In order to have a complete understanding of how well each scheduling algorithm works in the area of real-world applications, we select one application from each type. A loop with non-affinity and unpredictable workload is a rare case in practice. Therefore we only evaluate scheduling algorithms for applications of the first five types of loops.

6.4.2 Applications

The selected application kernels including potential affinity loops are Successive Over-Relaxation (SOR) (type I), Jacobi Iteration (JI) (type II) and Transitive Closure (TC) (type III). Matrix Multiplication (MM) (type IV), and Adjoint Convolution (AC) (type

V) are application kernels including non-affinity loops.

Type I: Balanced affinity loops in SOR.

```

DO SEQUENTIAL 1 I= 1, L
  DO PARALLEL 2 J= 1, N
    DO SEQUENTIAL 3 K= 1, N
      A(J,K) = UPDATE(A,J,K)
3    CONTINUE
2  CONTINUE
1  CONTINUE

```

All the iterations of the SOR parallel loop take about the same time to execute and each iteration always accesses the same set of data. Exploiting processor affinity may improve performance more than balancing the workload. In this application, each parallel iteration has locality rate of 1 and a data set of N array elements. The computational granularity of each parallel iteration is $O(N)$.

Type II: Predictable affinity loops in a Jacobi Iteration (JI)

```

DO SEQUENTIAL 1 I=1,L /* L controls iteration precision. */
  DO PARALLEL 2 J=1, N
    X1[J]=0
    DO SEQUENTIAL 3 K=1, N
      IF (A[J][K] .NE. 0).AND.(J .NE. K)
        X1[J]=X1[J]+A[J][K]*X0[K]
3    CONTINUE
    X1[J]=(B[J]-X1[J])/A[J][J]
2  CONTINUE
  DO SEQUENTIAL 4 L=1, N
    X0[L]=X1[L]

```



```

4      CONTINUE
1      CONTINUE

```

In the JI program, the top 20% of rows of elements in the non-singular matrix A are non-zero elements which are generated by a random number generator. The iterations of the parallel loop have a different workload which is determined by the distribution of non-zero elements in A , so exploiting load imbalance would improve performance. However, the workload of each parallel iteration is not changed when it is executed repeatedly.

The j -th iteration of the parallel loop always accesses the j -th row of the matrices A , $B[j]$ and $x0[j]$. When the j -th iteration is fixed to be executed repeatedly on a processor, it only needs to reload $x0[j]$ into a cache because $x0[j]$ is updated after each execution of the parallel loop. Hence, this application kernel exhibits good processor affinity. Each iteration has a data set of the size of $N+2$ elements, and data locality close to 1. The average computational granularity of each iteration is smaller than that in the SOR kernel.

Type III: Unpredictable imbalanced affinity loops in the Transitive Closure (TC) kernel.

```

DO SEQUENTIAL 1 I= 1, N
  DO PARALLEL 2 J= 1, N
    IF (A[J][I] .EQ. TRUE) THEN
      DO SEQUENTIAL 3 K= 1, N
        IF (A[I][K] .EQ. TRUE)
          A[J][K]=TRUE
3      CONTINUE
2    CONTINUE
1  CONTINUE

```

The TC program may exhibit more serious load imbalance than JI, where each iteration of the parallel loop and each execution of a parallel iteration may have computational granularity of $O(1)$ or $O(N)$, depending on the input matrix A . The iterations exhibit a weaker affinity to processors than SOR and JI. Due to the random computation feature, it is difficult to quantify the data locality and affinity of each parallel iteration.

Type IV: Balanced non-affinity loops in Matrix Multiplication (MM).

```

DO PARALLEL 1 I= 1, N
  DO PARALLEL 2 J= 1, N
    DO SEQUENTIAL 3 K= 1, N
      C[I] [J]=C[I] [J]+A[I] [K]*B[K] [J]
3      CONTINUE
2      CONTINUE
1      CONTINUE

```

The MM program does not have affinity to exploit. All the parallel iterations have computational granularity of $O(N)$. So, reducing synchronization and loop allocation overhead is the only way to improve performance. This application is used to investigate whether the adaptive algorithm has a lower scheduling overhead than the ML algorithm.

Type V: Predictable imbalanced non-affinity loops in Adjoint Convolution (AC).

```

DO PARALLEL 1 I=1, N*N
  DO SEQUENTIAL 2 K=I, N*N
    A[I]=A[I]+X*B[K]*C[I-K]
2      CONTINUE
1      CONTINUE

```

Similar to the matrix multiplication application, the parallel loop in the AC kernel only executes once, hence it does not exhibit processor affinity. However, the computational

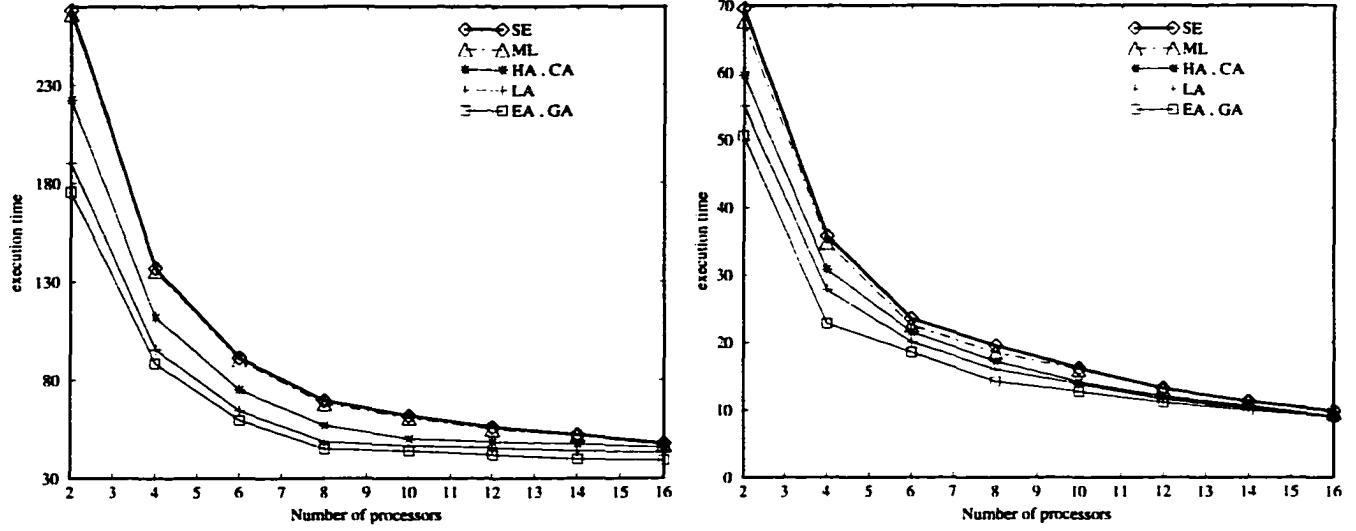


Figure 6.1: Performance of SOR on the KSR-1 (left figure) and Exemplar (right figure).

granularity of the i -th parallel iteration is $O(N^2 - i)$, changing as a specific function of the control variable i to produce a significant imbalanced load distribution (a triangular pattern). This kernel is used to examine how efficiently the adaptive algorithms can handle the load imbalance caused by a uniform partition.

6.5 Experimental Results

The performance metric we use to evaluate algorithms is execution time. Execution time measures how differently the scheduling algorithms work for different types of applications for given problem size.

6.5.1 Comparisons of Loop Scheduling Algorithms

First we use N/P^2 as the α value in our four adaptive scheduling variations CA, LA, EA and GA. We shall discuss the effect of the α value on performance in a later section, and discuss why $\alpha = N/P^2$ is cost-effective in the next section.

Figure 6.1 presents the execution time (in seconds) of SOR ($L=500$, $N=1024$)

running on 2 to 8 processors on both the KSR-1 and the Convex Exemplar. Since SOR is a perfectly balanced application kernel, the dynamic partition of the SE algorithm did not improve the performance of the ML affinity algorithm. On the other hand, it introduced some overhead into the ML algorithm. As the result, the ML and the SE perform the worst among them all, due to the overhead caused by more loop allocation and synchronization steps. By adaptively increasing the chunk size each time when a processor accesses the local work queue, the adaptive algorithms reduce the times that processors need for accessing the local work queues, therefore scheduling and synchronization overhead is reduced. All our five adaptive algorithms outperformed the ML and the SE algorithms. The EA and GA performed the best among them all, since they take no more than 3 steps to adjust their chunk size to finish the remaining iterations. The LA variation needs more allocation steps than the EA and the GA need. The HA and the CA variations change the chunk size in a limited range; therefore they could not get the best benefit by reducing the synchronization and loop allocation overhead for perfectly balanced applications.

Figure 6.2 plots the execution time of the Jacobi Iteration (JI) ($L=500$, $N=1024$) for the different scheduling algorithms on the KSR-1 and on the Convex Exemplar. JI should be an application that fits the SE algorithm best. Since the workload distribution illustrates a “rectangular” shape — the leftmost 20% having a very heavy load and the remaining 80% having almost zero workload. The SE algorithm can readjust the initial partition to balance the workload for each processor to improve the execution time. The lower execution time curves of the SE algorithm confirm this.

Instead of readjusting the initial partition, our adaptive algorithms reduced the execution time by adjusting the chunk size for each processor. A lightly loaded processor took a larger number of iterations to execute. Then it turned to help the heavily loaded processor. The heavily loaded processor took a small number of iterations to execute, and it might leave some iterations for the other processors to finish.

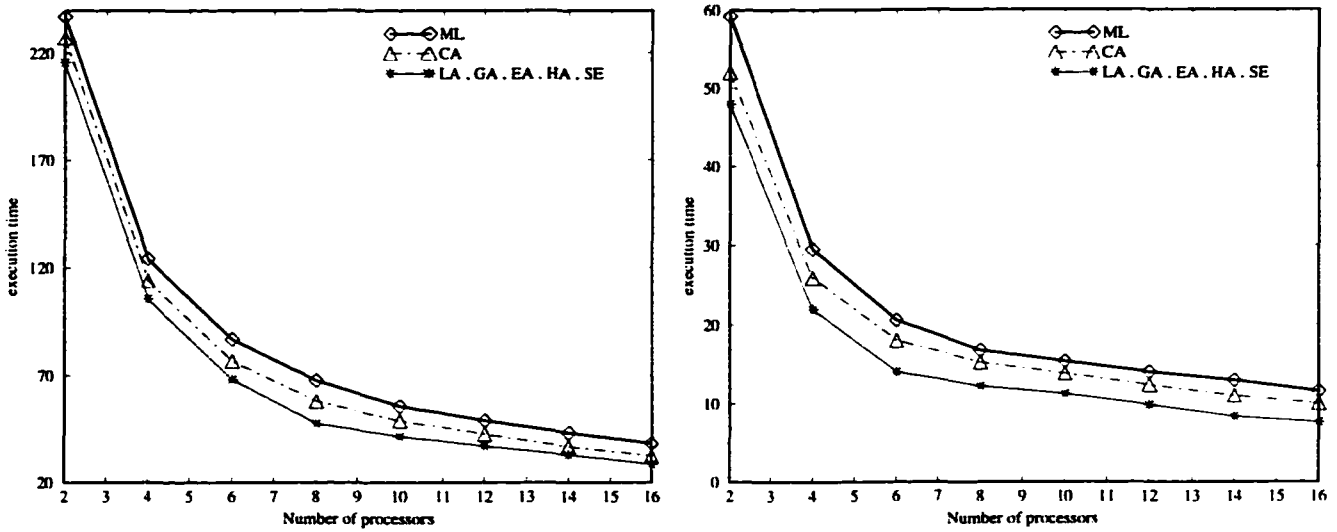


Figure 6.2: Performance of JI on the KSR-1 (left figure) and the Exemplar (right figure).

For solving a linear system of size $N=1024$ by the JI kernel, our LA, GA, EA, and HA variations perform as well as the SE. The CA variation performed slightly worse than the other adaptive algorithms because, when using the CA variation, the processors with zero workload still cannot take more than $2/P$ iterations to execute. Therefore they need more time to finish their lightly loaded jobs and turn to help the heavily loaded processor. In the meantime, the heavily loaded processor may have already taken a large number of jobs to execute and did not leave enough jobs for the idle processors.

Figure 6.3 presents the execution time of the transitive closure kernel with a random input graph of 1024 nodes, where about 10% of the edges are uniformly presented. In each execution of the parallel loop, the workload is uniformly distributed among iterations. However, the total workload increases at the next execution of the parallel loop. Figure 6.3 left and Figure 6.3 right show the comparative performance of seven tested algorithms respectively on both the KSR-1 machine and the Exemplar. The SE algorithm and the ML algorithm perform similarly because, in this case, the SE algorithm had little chance to improve the ML algorithm by readjusting the load distribution. Algorithms LA, EA, and GA performed the best among them all because they adjusted

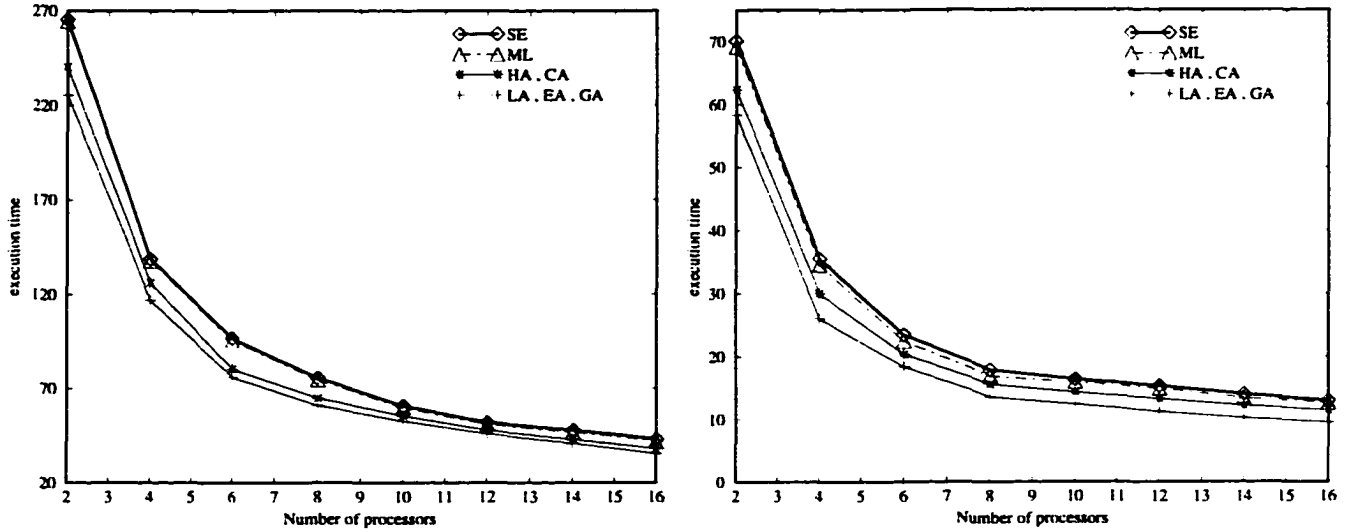


Figure 6.3: Performance of TC with random input on the KSR-1 (left figure) and the Exemplar (right figure).

scheduling granularity more aggressively. Combining these results with the experimental results of SOR, we conclude that for load balanced applications, aggressively adjusting scheduling granularity is an efficient method to reduce scheduling and synchronization overhead, thus to improve performance well. These results also show that the overhead of collecting state information is not significant compared with the benefit gained from adaptively adjusting scheduling granularity.

Again, we tested the scheduling algorithm and its variations for the transitive closure kernel with a skewed input graph of 640 nodes containing a clique of 320 nodes, and no other edges. In this case, load imbalance is significant in the computation across iterations and the total load of the parallel loop increases from one execution to the next. The execution times for each scheduling algorithm on the two platforms are presented in Figure 6.4(left) and Figure 6.4(right). Although the authors of the paper [68] claim that the SE algorithm assumes that the execution time of any particular iteration does not vary widely from one execution of the loop to another, our results show that the SE algorithm can still improve the ML algorithm in our case studies. Because our adaptive

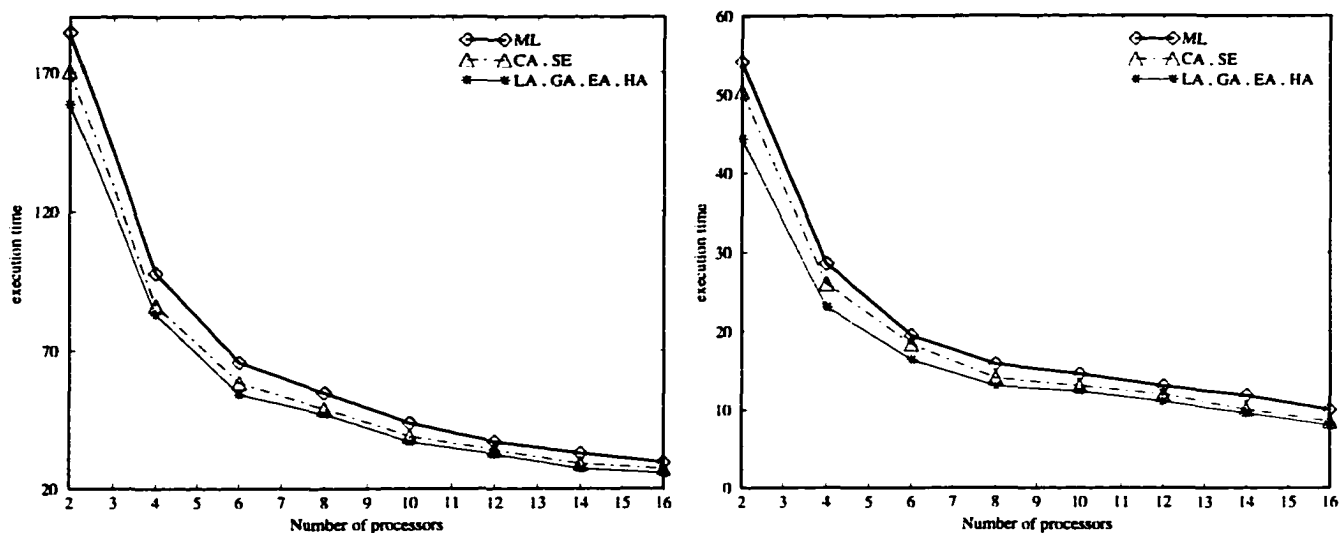


Figure 6.4: Performance of TC with skewed input on the KSR-1 (left figure) and the Exemplar (right figure).

algorithm captures the variance in load more precisely than the SE algorithm, LA, EA, GA and HA, performed better than the SE algorithm. The CA variation performed similarly to the SE algorithm. These experimental results show that adaptively adjusting scheduling granularity is an efficient way to handle the load imbalance in unpredictable loop applications.

If the parallel loop is not embedded in a sequential loop (we call this a non-affinity loop), both the SE algorithm and our heuristic variation HA have no chance to improve the ML affinity algorithm, because they adjust the initial partition or adjust the chunk size near the end of one execution of the parallel loop and hope that the new partition or the new chunk size can play a role in the next execution of the parallel loop. Now we want to see if other adaptive variations can perform better than the ML algorithm for the non-affinity loop.

Figure 6.5 left and Figure 6.5 right present the performance of the scheduling algorithms for the matrix multiplication (MM) with $N=512$. Algorithms ML, SE and HA performed similarly. Algorithms EA, LA, GA dynamically detect the workload distribu-

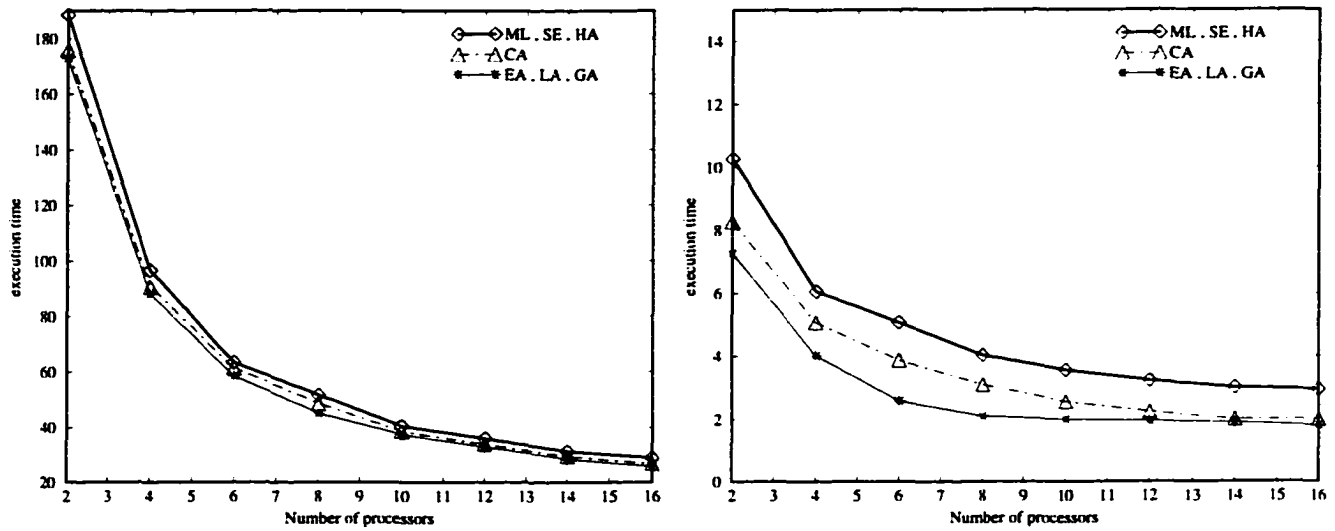


Figure 6.5: Performance of MM on the KSR-1 (left figure) and the Exemplar (right figure).

tion conditions and rapidly increase the chunk size, so that the processors take all the remaining iterations to execute after only a few accesses to the local work queue. The CA variation also increases the chunk size to a limit ($2/P$ of the remaining iterations); therefore it involves less synchronization and loop allocation overhead than ML but presents more overhead than GA, LA, and EA. Compared with the experimental results on kernel SOR, adaptive variations do not improve the ML algorithm on MM significantly because the parallel loop only executes for one time.

Figure 6.6 left and Figure 6.6 right present the performance of the scheduling algorithms for kernel Adjoint Convolution with $N=128$. SE and HA could not improve the ML algorithm, since the parallel loop is not embedded within a sequential loop. Load imbalance across iterations was significant since the first iteration took time proportional to $O(N^2)$, while the last iteration took time proportional to $O(1)$. As expected, ML, SE and HA performed similarly, while EA, LA, and GA performed the best among them all. The CA variation's performance was in between.

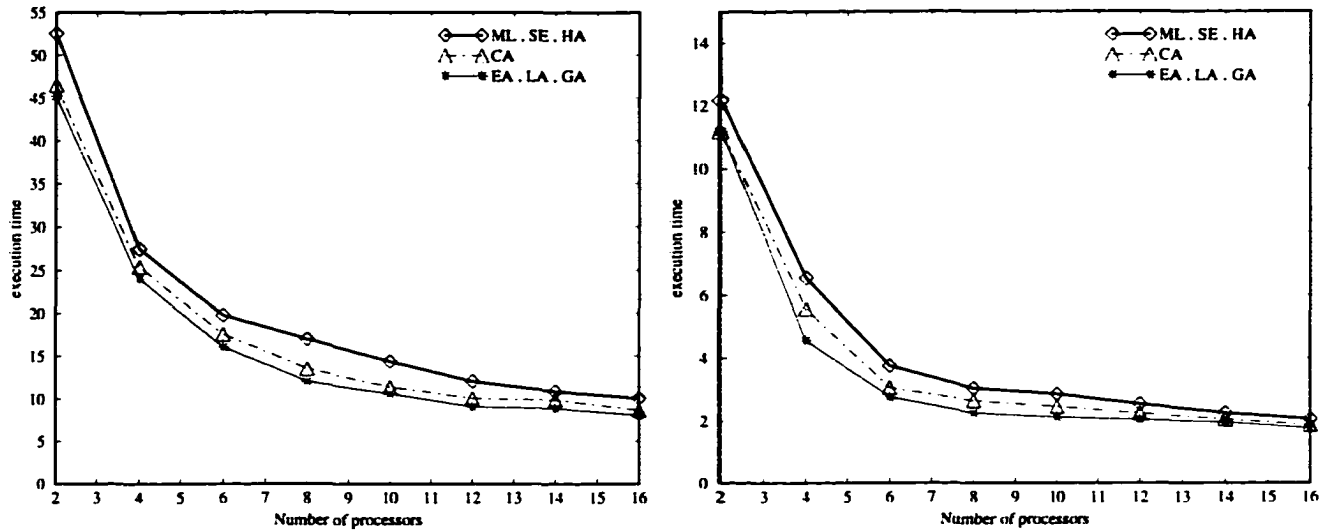


Figure 6.6: Performance of AC on the KSR-1 (left figure) and the Exemplar (right figure).

6.5.2 Determine the Cost-Efficient Value

In the previous section, we used N/P^2 as the value of α in our adaptive scheduling algorithm and its variations, where N is the number of iterations in the parallel loop and P is the number of processors we used to execute the parallel loop. Here we tested several values of α in, trying to give an optimal value.

We evaluated our adaptive scheduling algorithm and its variations with different α values for the five benchmark applications on both the KSR-1 and the Exemplar. The α values we selected to evaluate are N/P , N/P^2 , 32 and 4 respectively. Due to space limitation, we only present part of the results here for two of our adaptive scheduling variations EA and CA with respect to one kernel application. The remaining results that we do not specify in what follows also support the conclusions we are going to present.

Figure 6.7 left presents the performance of the SOR kernel on the KSR-1 using the EA adaptive variation with the different α values. We also present the performance of the ML algorithm running the SOR kernel for a comparison. EA with $\alpha = N/P$ and $\alpha = N/P^2$ showed the best performance, while EA with $\alpha = 4$ presented the worst

performance. Since SOR is a well balanced application, all the processors should have a normal workload. A very large value of α like N/P guarantees that the workload state of each processor is always “normal” so that the processor can increase its chunk size and reduce its execution time. Although SOR is well balanced, sometimes events such as cache misses, page faults, and interprocessor communication delays can cause some execution time variance among iterations. If we use a very small value for α , such as $\alpha = 4$, in the presence of interference from such kinds of events, some processors take their workload states as “heavy” and therefore decrease their chunk size by a factor of two. Since we do not give a limit for the chunk size for the EA variation, this decrease of chunk size at an exponential rate may cause some processors to take a very small chunk so that the processor may take only one iteration for each access to the local work queue (similar to *self-scheduling*). This is why the EA (with $\alpha = 4$) spent much more time than the ML when the number of processors is 2 or 4. When the number of processors increases to 6 and 8, $\alpha = 4$ becomes close to N/P^2 . The same reason holds with $\alpha = 32$ for EA. When $P = 2$, the processors cannot determine their workload states correctly due to the system interference and the small value of α . As the number of processors increases, the $\alpha = 32$ gets close to the value of N/P^2 . Therefore the algorithms performed well with number of processors 4, 6 and 8.

Figure 6.7 right presents the performance of the SOR application on the KSR-1 using the CA adaptive variation with different α values. We also show the curve for the ML algorithm and the curve for the EA variation with $\alpha = 4$ in order to compare them. Figure 6.7 right shows that CA with $\alpha = N/P$ and $\alpha = N/P^2$ performed the best among them all. CA with $\alpha = 4$ shows that too small a value of α (in comparison with the value of N/P^2) may cause a negative effect on the performance of our adaptive algorithm due to system interference. CA with $\alpha = 4$ performed the worst among the other CA curves. But we also notice that this curve is much lower than that of the EA with $\alpha = 4$. The reason is that we limit the range of chunk size for the CA variation within $[P/2, 2P]$. It

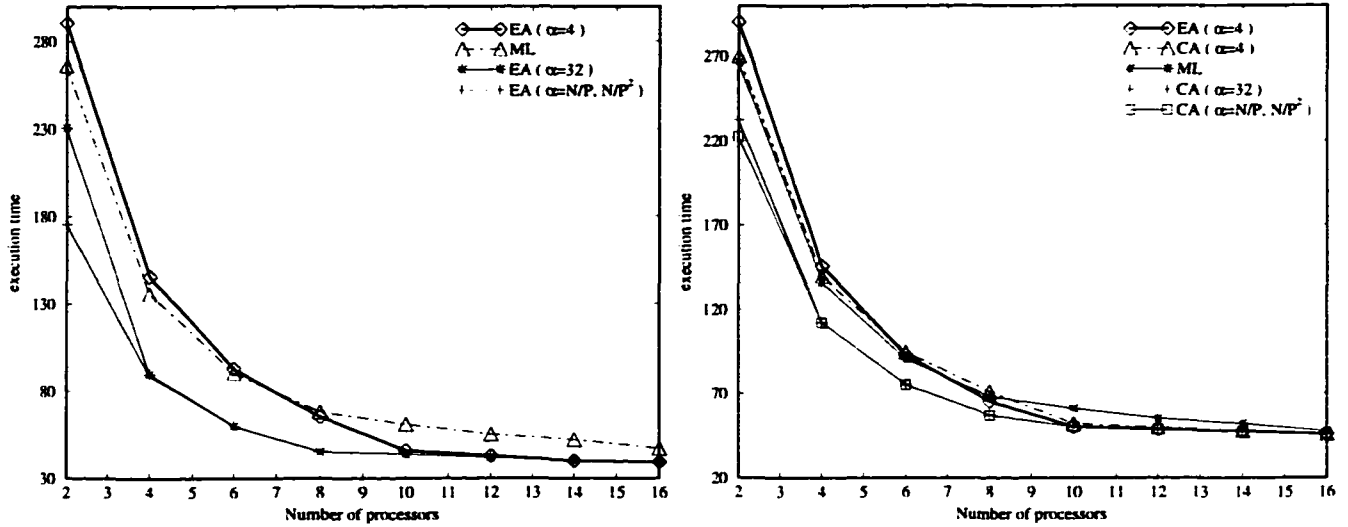


Figure 6.7: Performance of SOR on the KSR-1: (a) using EA with different α values (left figure); (b) using CA with different α values (right figure)

guarantees that the processor takes at least $\frac{1}{2P}$ of the remaining iterations to execute each access to the local work queue.

Figure 6.8 left and Figure 6.8 right plot the performance of the Jacobi Iteration (JI) on the Exemplar using the EA and CA adaptive variations with different α values. Again, this kernel presents highly unbalanced workload. A large α value makes all the processors think they are having the “normal” workload, thus to increase the chunk size and to get more and more iterations to execute. As a result, the heavily loaded processor has to finish almost all the iterations by itself, leaving no iterations for the lightly loaded processor to execute. That is why the performance of EA and CA with $\alpha = N/P$ was poor (Figure 6.8 left and Figure 6.8 right). On the other hand, since CA limits the chunk size within a range, the performance of CA with $\alpha = N/P$ was better than the one of EA with $\alpha = N/P$ (Figure 6.8 right).

Figure 6.8 left and Figure 6.8 right also show that the cost-efficient value of α is N/P^2 . A small value of α (in comparison with the value of N/P^2) may not give precise information to the processors about their workload state; therefore it would affect

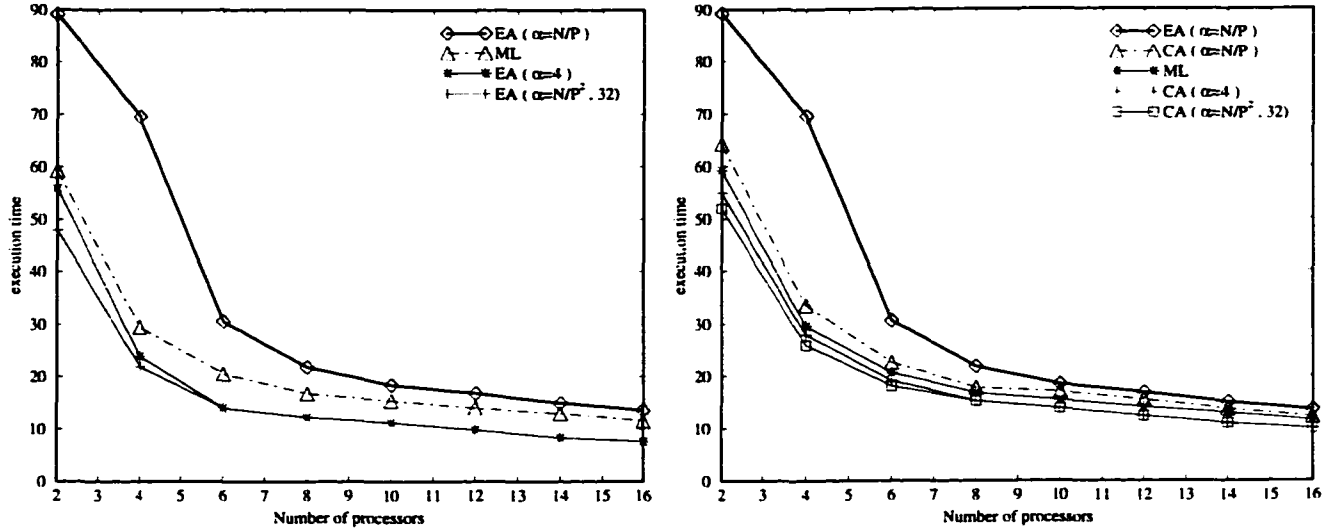


Figure 6.8: Performance of JI on the KSR-1: (a) using EA with different α values (left figure); (b) using CA with different α values (right figure)

execution performance. This can be seen from the curves of EA and CA with $\alpha = 4$. When $\alpha = 32$, which is close to the value N/P^2 , EA and CA also presented reasonably good performance for running this kernel.

6.5.3 Summary of Comparisons

By adaptively adjusting the loop allocation granularity according to the workload and execution speed of each processor, our loop scheduling algorithm demonstrated better performance than the affinity scheduling algorithm proposed by Markatos and Leblanc in [51] and the dynamic partitioned affinity scheduling algorithm proposed by Subramaniam and Eager in [68]. These authors had showed that the two algorithms presented the best performance among all the loop scheduling algorithms. Our adaptive scheduling algorithm is suitable for a wider range of application programs. The algorithm can reduce the execution time not only for well load-balanced parallel loops, but also for those load unbalanced parallel loops. Our experiments show that the overhead caused

by collecting state information is not significant compared with the benefit gained. One important conclusion from this research is that efficiently using runtime information can significantly improve the efficiency of loop scheduling algorithms.

Among the variations of the adaptive scheduling algorithm, the EA, LA and GA variations always demonstrate better performance than the CA and HA variations. Although EA, LA, and GA have higher risk than CA in terms of causing load-imbalance, and in terms of being much more sensitive to system interference, we have not observed the worst performance phenomena in our case studies, such as the Ping Pong effect where the state of a processor is often switched between the lightly loaded and the heavily loaded to cause overwhelming scheduling overhead. In addition, the negative effect of the EA, LA and GA variations can be significantly reduced by selecting the appropriate workload control constant α as N/P^2 .

Machine architecture may be another important factor that affects the performance of loop scheduling algorithms. So far, we have been able to test our adaptive algorithm and its variations only on the KSR-1 and the Exemplar. Our experimental results indicate that the algorithm's performance is quite independent of shared-memory architectures. However, the effectiveness of the adaptive algorithm is significantly affected by the system size. When the system size scales very large, the cost to collect runtime information increases so that the advantages of the adaptive algorithm are nullified by the increased overhead. So, the adaptive algorithm is very suitable for scheduling parallel loops over a small number of processors.

6.6 Locality-preserved Task Scheduling

In our run-time system, task scheduling takes into consideration the exploited locality among tasks. The tasks in a partition are grouped in task bins based on their data-access affinities. The scheduling algorithm must take advantage of the exploited locality while

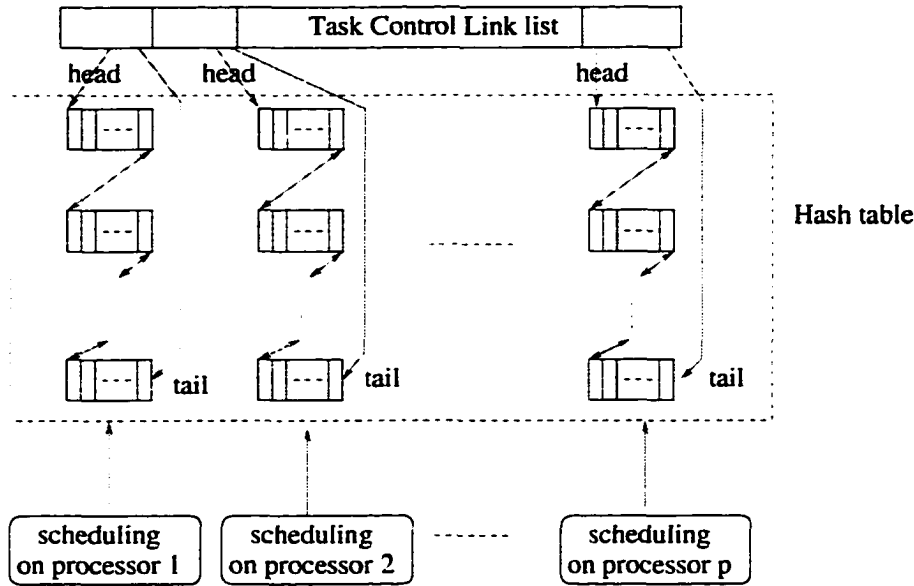


Figure 6.9: Scheduling framework.

minimizing load imbalance, because both locality and load imbalance have significant impact on performance. The proposed scheduling algorithms in section 6.3 cannot be directly applied. Here, we extend the linearly adaptive algorithm to schedule the tasks in the partitions generated from the locality optimization phase. The extended algorithm is called the Locality-preserved Adaptive Scheduling algorithm, denoted as LAS. Because the number of processors in the targeted SMP system is in the range of small scale to medium scale, the linearly adaptive algorithm is aggressive enough to reduce synchronization cost.

From the scheduling point of view, tasks in the multi-dimensional hash structure are organized in several separate task queues as shown in Figure 6.9. The head and the tail of each task queue are pointed to by a corresponding record in the Task Control List (TCL). Let p be the number of processors. Initially, the i -th task group chain in the TCL list is assigned as the local task chain on processor i , for $i = 1, 2, \dots, p$. This initial allocation maintains the minimized data sharing achieved in the task reorganization step

among processors. The number of tasks to be processed on processor i is counted by the corresponding TCL counter variable, denoted C_i , which is used in the LAS algorithm to estimate load imbalance, just like the processor speed variables do in the adaptive algorithm given in section 6.3. However, the TCL counter variables are different from the processor speed variables. A processor speed variable records the number of tasks that have been finished in the corresponding processor. This allows the processor to estimate precisely how many tasks remain, because the tasks are evenly partitioned among processors initially. It does not work in the run-time system because the task chains in the TCL list may contain imbalanced numbers of tasks. In addition, each processor has a chunk control variable with initial value of p , denoted K_i for processor i , to determine how many tasks to be executed at each scheduling step.

The LAS algorithm still works in two phases: the local scheduling phase and the global scheduling phase. However, the head of a task queue can only be accessed by its local processor and the tail of the queue is accessed by remote processors. This restriction tries to reduce the number of task groups whose tasks are split onto different processors to execute. By doing so, the LAS algorithm also attempts to take advantage of the exploited locality while achieving a good load balance. All the processors start at the local scheduling phase. The algorithm is described for processor i ($i = 1, 2, \dots, p$) as follows.

Local scheduling : Processor i first calculates its load status relative to the other processors as follows:

$$\text{heavily loaded} \quad \text{if } C_i > \sum_{j=1}^p C_j / p + \alpha \quad (6.2)$$

$$\text{lightly loaded} \quad \text{if } C_i < \sum_{j=1}^p C_j / p - \alpha \quad (6.3)$$

$$\text{normally loaded} \quad \text{otherwise} \quad (6.4)$$

Here, α is an adjustable parameter. In the experiments, we selected $\alpha = N/p^2$ (N is the number of tasks or iterations).

During the above computation, if the number of remaining tasks in one processor's local chain is found to be 0, i.e., $\exists_{j \in [1, p]} (C_j = 0)$, processor i sets its chunk control variable, K_i , to p , then goes to the global scheduling phase. Otherwise, processor i linearly adjusts its chunk control variable according to its load status as follows:

$$K_i = \begin{cases} \max\{p/2, K_i - 1\} & \text{if its load is light} \\ \min\{2p, K_i + 1\} & \text{if its load is heavy} \\ K_i & \text{otherwise} \end{cases} \quad (6.5)$$

The varying range $[p/2, 2p]$ for the chunk control variables has been shown to be safe for balancing the load [51, 61]. Then processor i gets the C_i/K_i tasks from the head of its local task queue. Having finished the allocated tasks, processor i goes back to repeat the local scheduling.

Global scheduling : First, processor i always gets C_i/K_i tasks from the head of its local task chain to execute until its local task chain is empty. Then, processor i gets $1/K_i$ of the remaining tasks from the tail of the local task queue in the most heavily loaded processor until all the processors empty their local task queues.

In the local scheduling phase of the LAS algorithm, tasks are executed in group-by-group order where the tasks in a group are always executed together on the local processor. Only when a processor is trying to help other processors in the global phase may tasks in a group split to execute on different processors. This could possibly weaken the optimized locality. In the global scheduling phase, emphasis is put on load balancing. Usually, the global scheduling phase is executed for a short period in comparison with the local scheduling phase, provided that the partitions generated in the task reorganization step are not poorly imbalanced. The LAS algorithm maintains the feature of exploiting processor affinity while tasks are repeatedly executed. Our experimental results presented in the next chapter show its effectiveness.

Chapter 7

Performance Evaluation

This chapter describes the goals, the methods, and the environments of our performance evaluation. Then, the performance results are reported.

7.1 The Goals

As we described in section 1.1, the design of our run-time technique for optimizing cache locality is motivated by addressing difficult issues in improving the memory performance of the applications with dynamic memory-access patterns. So, we first must show the effectiveness of our run-time technique with respect to this type of application.

Compiler-based locality optimizations have been shown to be able to improve the memory performance of applications with regular computation patterns and memory-access patterns (as overviewed in section 2.5) significantly. However, compiler-based locality optimizations have not been widely available in commercial SMP systems. Our run-time system was implemented as a set of run-time library functions, which can be directly used by users. So, the second evaluation goal is to study how efficiently the run-time technique works in comparison with compiler-based techniques with respect to

regular applications.

The third evaluation goal is to investigate the behavior of the run-time technique. This includes two aspects: (1) the effect of run-time locality optimizations on the memory-access pattern of a program, and (2) the effects of various performance factors on the quality of the run-time technique.

7.2 Evaluation Method

In performance evaluation, three approaches are usually applied: modeling, simulation and measurement [91, 89]. A convincing performance evaluation must use at least two approaches to verify performance results. As described in Chapter 3, it is difficult to model the cache locality optimization problem on SMPs precisely. So, we use both the simulation approach and the measurement approach to evaluate our run-time system. The major advantage of the simulation approach over the measurement approach is that the simulation can provide detailed information on the execution behaviors of a new technique. An insightful analysis can be conducted using simulation. A major drawback of the simulation is that a simulator cannot be built to reflect a real system exactly when the system is very complex, such as an SMP system. So, measurement should be used to investigate the practical performance of a technique in real experimental environments.

Using a detailed simulator we built, we studied the effectiveness of the run-time system in exploiting the cache locality of applications with respect to the changes of the cache miss rate, bus traffic, execution time, and cache interference. Then, we further measured the effectiveness of the run-time technique on two commercial SMP systems. Besides the concrete evaluation approaches, the selection of performance metrics and benchmarks is important for performance evaluation.

7.2.1 Performance Metrics

Locality optimizations are aimed at reducing the number of cache misses. As we discussed in section 2.3, there are different types of cache misses that can be targeted for reduction by a locality optimization technique. The reduction pattern in different types of cache misses provides an insightful view on the effectiveness of a locality optimization technique. So, **cache-miss reduction** is our first performance metric. In general, a finer classification of cache misses requires a larger simulation time. We refine cache-miss types to an appropriate level so that an application can be simulated in a reasonable time without losing the precision. This enables us to study the effectiveness of our run-time technique in increasing the total number of data reuses in caches and in reducing the total number of shared data among caches.

Cache misses are classified as the following three types:

- **Compulsory misses:** misses caused by reads or writes on data that have never been brought into the cache before.
- **Replacement misses:** misses caused by reads or writes on data that were brought into the cache but replaced by other block data most recently.
- **Coherence misses:** misses caused by reads or writes on data that were brought into the cache but invalidated by other processors most recently.

The reduction in the total number of the first two types of misses measures the effectiveness of the task regrouping technique proposed in section 5.2. The last type of misses is caused by data sharing among multiprocessors. However, the total number of coherence misses cannot precisely reflect the data sharing. In the example as shown in Figure 7.1, a processor consecutively executes three operations $r(a) \rightarrow r(b) \rightarrow r(a)$. Assume that data *a* and *b* be mapped onto the same cache block and an invalidation on the cache block happens between $r(a)$ and $r(b)$. In this pattern, the miss caused by the second $r(a)$ is

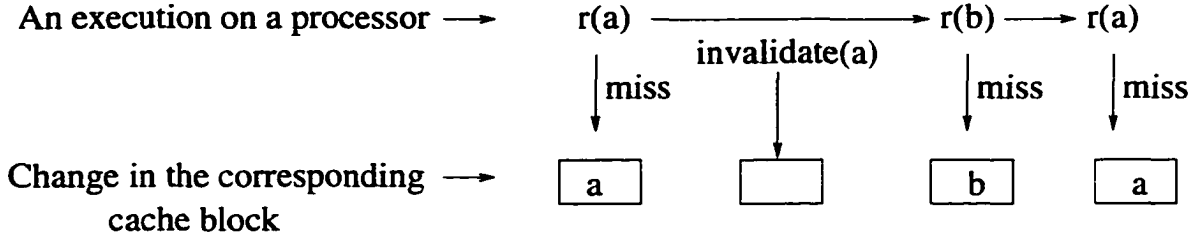


Figure 7.1: The cache-access pattern of a simply execution sequence on a processor where data a and b are mapped into the same cache block. $r(a)$ expresses a read on data a .

counted as a replacement miss, because no matter whether $\text{invalidate}(a)$ happens, $r(b)$ will replace a with b . So, the invalidation caused by data sharing has not been counted into the number of coherence misses. In order to measure data sharing more precisely, we counted the total number of invalidations.

In order to investigate the network contention, the second metric we used is **communication size**, which is the total amount of data communicated. In a SMP system, there are three types of communications: cache-to-cache, cache-to-memory, and memory-to-cache.

Regarding the execution performance of the run-time technique, three metrics are used:

1. Execution time, which measures the overall performance of a computation.
2. Balance coefficient, denoted as $\text{Imbalance}(C)$ for computation C , which is defined as follows:

$$\text{Imbalance}(c) = \frac{Dvi(C)}{\bar{T}}, \quad (7.1)$$

where $Dvi(C)$ and \bar{T} are, respectively, the standard deviation and the mean of the execution times of the multiprocessors used in a parallel computation. A smaller value of Imbalance refers to a computation with a better load balance. When $\text{Imbalance} = 0$, Dvi is 0 where all processors have the same execution time.

```

int b = n/p; /* here matrix size n is assumed to be evenly divided
              by the number of processors. */

for (kk=0; kk<n; kk+=bf)
  for (jj=0; jj<n; jj+=bf)
    for (i=pid*b; i<(pid+1)*b; i++) /* task partition */
      for (j=jj; j<min(jj+bf, n); j++){
        d = A[i][j]; /* d is register type */
        for (k=kk; k<min(kk+bf, n); kk++)
          d += B[i][k] * C[j][k];
        A[i][j] = d;
      }

```

Figure 7.2: A well-tuned parallel version of the DMM application [82]. Here, `pid` is a thread id of value from 0 to $p-1$. p is the number of threads.

3. Overhead percentage, which is the rate of run-time overhead to execution time.

This is used to measure the implementation efficiency of our run-time system.

7.2.2 Benchmarks

In section 4.3.2, applications have been classified into four types. The three benchmarks, DMM, AC, and SMM, have covered all the application types that fit our programming model. We evaluate the performance of our run-time system using these three benchmarks. The optimized versions of DMM, AC and SMM using our run-time library functions, which are given in section 4.3.2, are denoted as `DMM_LO`, `AC_LO`, and `SMM_LO`, respectively.

For comparison, the three benchmarks are parallelized respectively using the best existing techniques as follows.

1. We transform the dense matrix-matrix multiplication program shown on the left side of Figure 4.4 into the sequential block algorithm proposed by Wolf and Lam [82]. Then, the transformed program is parallelized by uniformly partitioning computations on multiple processors, shown in Figure 7.2. The parallelized version,

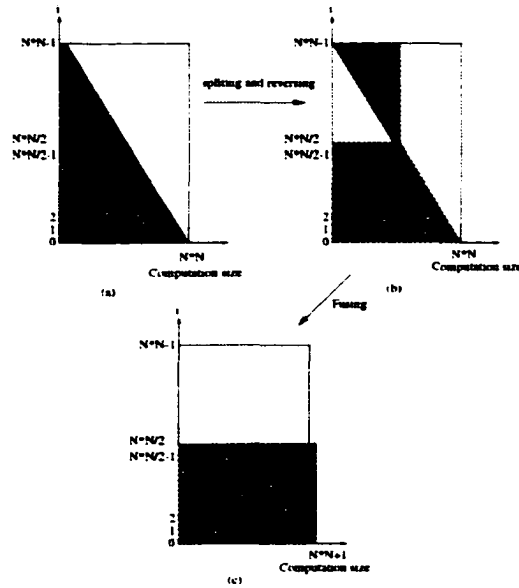


Figure 7.3: Optimizing locality and balance of AC.

denoted as DMM_WL, has well-optimized cache locality and perfect load balance. The locality of the block algorithm is further improved by transposing one matrix so that the innermost loop accesses contiguous memory regions on the two arrays. Based on this benchmark which can be well optimized by compiler-based optimizations, we are interested in investigating whether the run-time technique can achieve as competitive performance as that by compiler-based optimizations.

2. For the adjoint convolution program shown on the left side of Figure 4.6, each iteration of the outermost loop accesses a contiguous segment of arrays A and C respectively. In order to investigate the effect of locality optimization, we assume that arrays A and C are too large to fill a cache. Two iterations of the outer loop that have closer values of index i have larger overlap between their data sets. From the standpoint of optimizing cache locality, the AC program should be parallelized by using the blocking technique to chunk the outermost loop. By this, each processor will be allocated with a set of adjacent outermost iterations. Because the iterations

of the outermost loop have decreasing workload as index i decreases, a varying-sized blocking technique should be used to optimize both locality and load balance. For any given N and the number of processors, it is difficult or impossible to choose a set of different block sizes to balance load among processors.

Here, we integrate several compiler optimizations to solve this problem. For the sequential program shown on the left side of Figure 4.6, the outer loop is a parallel loop where its iterations are data-independent and are called parallel iterations of the AC program. The computation pattern of the outer loop is visualized in Figure 7.3(a), where the vertical axis gives the loop indices of parallel iterations and the horizontal axis represents the computation sizes of parallel iterations. The computation pattern has an imbalanced triangular shape. The best performance of the AC program can only be achieved by optimizing both locality and load balance.

We first equally split the outermost loop into two loops and reversed the computations of the second loop. The changed computation pattern is shown in Figure 7.3(b) and the corresponding program is shown in Figure 7.4(b). Then, the second loop was aligned and fused with the first loop to make a new loop with balanced iterations. The program transformations are shown in Figures 7.4(c) and 7.4(d). The computation pattern of the final transformed program is shown in Figure 7.3(c), which has perfect load balance. Then, the final program is equally blocked onto multiple processors to maintain both load balance and cache locality.

3. The SMM program has an irregular memory-access pattern that is determined by run-time input data. This type of application is very hard for compiler-based techniques to partition effectively for cache locality optimization. For this kind of application, existing approaches to optimize the performance use run-time scheduling to minimize load imbalance and to exploit a kind of weak locality: processor affinity [49, 51, 61]. In section 6, we have proposed several more efficient schedul-

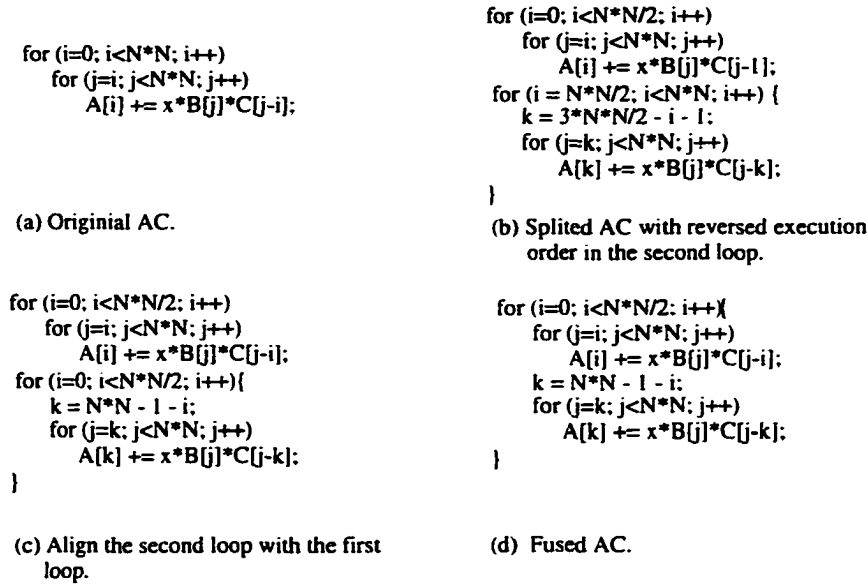


Figure 7.4: A well-tuned parallel version of the AC application.

ing algorithms which have been shown to outperform previous run-time scheduling techniques for imbalanced parallel loops. For comparison, we use the linearly adaptive scheduling technique to schedule the executions of parallel iterations in SMM because the scheduling algorithm in our run-time system is also derived from the linearly adaptive scheduling technique. Hereafter, we denote this parallelized version as SMM_A.

Although SMM_A has a similar execution procedure to SMM_LO, three significant differences are: (1) Initially, SMM_LO groups and partitions tasks with the objective of minimizing data sharing among partitions and maximizing data reuse in a partition. The SMM_A just cyclically puts tasks in local queues of processors. The SMM_LO has higher run-time scheduling overhead than that of SMM_A. (2) Although both SMM_LO and SMM_A use the linearly adaptive scheduling algorithm, the scheduling in SMM_LO is locality-oriented, which has a better chance to reduce its number of memory accesses. Compared with SMM_A, SMM_LO is

expected to further reduce execution time by optimizing memory performance on modern computers.

The last issue is how to select problem sizes of the tested programs. Because our goal is to evaluate the effectiveness of the run-time system in exploiting cache locality, we select the problem sizes based on the underlying cache size so that the data set of an application is too large for the cache to hold the whole data set. In simulation, in order to shorten simulation time without losing the confidence of simulation results, we selected relatively small problem sizes for the applications. We scale down the cache size accordingly for these programs. These selections can prevent the advantage of the run-time system from being shadowed by hardware cache, because on a given commercial system, cache capacity is fixed but the problem size of an application can be changed.

7.3 Performance Evaluation Environments

7.3.1 Event-Driven Simulator

The simulation was conducted on an event-driven simulator for bus-based shared memory systems, which was built on MINT, a MIPS interpreter [79]. The MINT software package simulates the execution of standard Unix executable files compiled for a MIPS R3000-based multiprocessor and generates events for specific instructions, such as read and write. The MINT simulator is a simulator for shared-memory machines where processors are simulated by processes. The MINT simulator provides a flexible interface for developing complicated back-end simulators for different multiprocessor architectures. We built a detailed simulator for the memory hierarchy of a bus-based SMP system.

Figure 7.5 shows the architectures and the interfaces of the MINT simulator and the memory-hierarchy simulator. A threaded program in C language is first compiled as an executable on multiple SGI-R3000 processors, which is then executed on the MINT

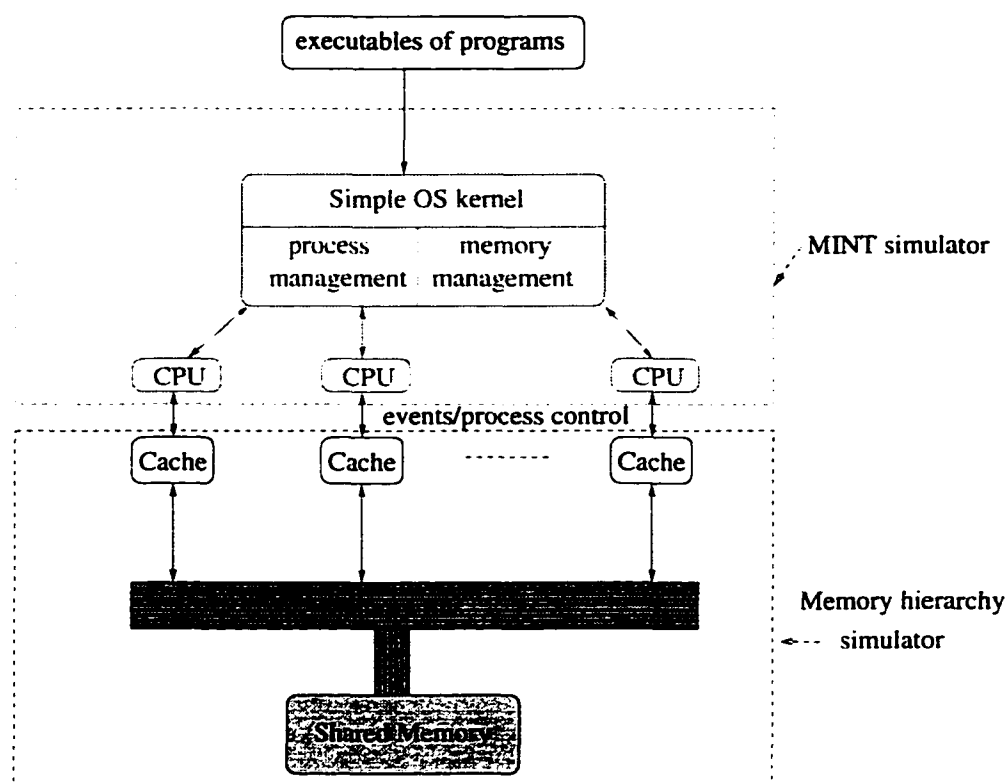


Figure 7.5: The architecture and the interfaces of the MINT simulator and the memory-hierarchy simulator.

simulator. The MINT simulator interprets the instructions of an executable to simulate instruction executions on multiple CPUs. Meanwhile, the MINT simulator can generate a set of events to drive the memory-hierarchy simulator, which simulates the memory-access procedure of memory-access operations. The memory-hierarchy simulator can change and monitor the execution of the MINT simulator through a process-control interface provided by MINT.

Because MINT only simulates non-pipelined processors, we implemented the sequentially consistent shared-memory model [1], where each processor only issues one memory access at a time and subsequent memory accesses stall until an issued memory access finishes. The simulated memory hierarchy has two levels: one cache level and one

shared-memory level. Although many commercial computers have multiple cache levels, one cache level is sufficient to study the effectiveness of the run-time system for exploiting the cache locality. Each cache block in the simulated cache has the following three state bits, denoted as MSI bits [19]:

Invalid bit , which indicates whether the data in a cache block has been invalid.

Shared bit , which indicates whether there are multiple copies in the system of the data held by the cache block.

Modified bit , which indicates whether the data held by the cache block has been changed since it was brought into the cache block.

In a multiprocessor, a set of the cache blocks with the same cache set address in multiple caches is called a cache line. When shared data is read by multiple processors, multiple copies of the data may exist in the cache blocks of different processors. But, all the copies must be in the same cache line. Our memory-hierarchy simulator simulates a memory-bus based SMP. In order to maintain the consistency of the shared data, a snooping cache coherence protocol was implemented. The implemented snooping protocol is described as follows, which is similar to the standard snooping protocol [30]:

1. Regarding a read from the local cache:

- (a) If it is a read hit, then return;
- (b) Otherwise, it is a read miss:

If the target cache block of the read is dirty but not shared, the *writeback* protocol [30] is used to save the target cache block into memory. Then a memory access request is put onto the memory bus. If there is a copy of the requested data in another cache, the whole cache block, including its state bits, is copied into the requesting cache and the corresponding cache line is

set to be shared; otherwise, the requested data is copied from memory where the state bits of the target cache block are set to be non-shared, valid, and non-modified.

2. Regarding a write to the local cache:

- (a) If it is a write hit, data is written into the local cache and an invalidation is broadcast to the other caches to invalidate the data copies. The target cache block of the data is set to be non-shared and modified.

- (b) Otherwise, it is a write miss:

An invalidation message is broadcast to the other caches to invalidate the data copies. Then data is written into memory and a copy is moved into the target cache block in the local cache. This is called the *written allocation* protocol [30] in which the data will be written to memory if the target cache block originally contains valid and modified data. The state bits of the updated cache block are set to be non-shared, valid, and non-modified.

Each instruction is assumed to take 1 cycle of execution time. To avoid overemphasizing the effect of cache misses in total execution time, we assume that the cache has a 1 cycle hit time, just like an on-chip cache, which helps to make a balance between the effect of instruction execution and the effect of memory accesses. To reduce artificial limitations as much as possible, we select simulation parameters for the bus and the memory system so that they are very close to modern commercial systems architecturally [14, 24, 66]. The shared main memory is fully pipelined, which has a memory bus of 64 bits and access latency of 24 cycles. The interconnection bus has arbitration time of 2 cycles, invalidation time of 3 cycles, and cache-to-cache transferring time of 3 cycles. The hardware synchronization primitive, `test&set`, is simulated to provide two high-level lock primitives: `mutex_lock` and `mutex_unlock`, which are used in the dynamic

schedulers.

7.3.2 Measurement environments

The HP/Convex S-class [7] is a crossbar-based cache coherent SMP system with 16 processors, while the Sun Hyper-SPARCstation-20 is a bus-based cache coherent SMP system with 4 processors. The architectural differences of these two SMP systems provide the run-time system with different opportunities/challenges to improve the performance of applications.

The HP/Convex S-class has 16 PA8000 processors of 720 peak MFLOPS. A PA8000 is a 4-way super-scalar RISC processor, supporting a 64-bit virtual address space, which operates at 180MHz. A PA8000 processor has a single level primary cache with separate instruction cache and data cache of size 1 MB each. The cache is direct-mapped using a write back policy, which has cache line size of 32B and cache hit time of 3 cycles (about 16.7 nanoseconds). Cache coherence is maintained by a distributed directory-based hardware cache coherence protocol. The S-class has a pipelined, 32-way interleaved shared memory of 8 memory boards, which are interconnected with processors by a 8×8 nonblocking crossbar. The data path from the crossbar to the memory controller is 64-bits wide and operates at 120 MHz. The access latency of a 32B cache line from the shared memory to a cache is 509 nanoseconds. The ratio between cache miss time and cache hit time is about 30.

Our Sun Hyper-SPARCstation-20 has 4 hyperSPARC processors operating at 100MHz. Each processor has a two-level cache hierarchy: a 64KB on-chip cache and a 256 KB virtual secondary cache where the cache line size is 64B. Compared with the S-class, the larger cache line of the Hyper-SPARCstation-20 exploits better spatial locality for applications. The cache hit time is about 300 nanoseconds. A cache miss time is about 13360 nanoseconds. The ratio of cache miss time to cache hit time is about 36.

Processors	Miss rate					
	DMM application		AC application		SMM application	
	DMM_WL	DMM_LO	AC_BF	AC_LO	SMM_A	SMM_LO
2	0.006	0.008	0.051	0.043	0.025	0.011
4	0.006	0.008	0.051	0.044	0.025	0.011
8	0.005	0.007	0.052	0.044	0.025	0.012

Table 7.1: Cache miss-rate based comparison where experiments were conducted under shrinking factor $f = 1$.

Cache coherence is maintained by the bus-based snooping protocol.

Compared with HP/Convex S-class with respect to instruction issuing rate and memory access latency, the Sun Hyper-SPARCstation-20 is much slower. In measurement, we focused on the comparison of relative performance results.

7.4 Performance Results

7.4.1 Simulation results

The selected array sizes of programs DMM, AC and SMM are 128×128 , 4096, and 256×256 respectively, which correspond to working sizes 384K, 96K, and about 450K, respectively. The simulation was conducted on a 16K cache with a 16B block size.

A. Cache Performance

Table 7.1 presents the miss rates of the six benchmark programs selected in section 7.2.2 on 2 processors to 8 processors. The distribution patterns of different types of misses in the programs are presented in Figures 7.6, 7.7, and 7.8. From the miss rates, we can

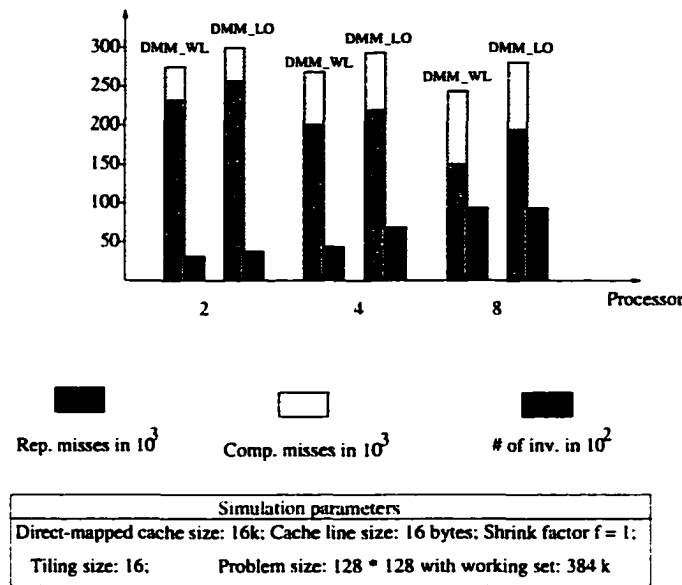


Figure 7.6: Cache performance comparison between DMM_WL and DMM_LO.

see that the miss rate is not much changed with the number of processors. However, the number of processors has a significant impact on the distribution patterns of different types of misses.

Regarding regular application DMM, the locality-optimized version (DMM_LO) using the run-time technique is 9% to 14% higher than the well-tuned version (DMM_WL) in the number of cache misses (Table 7.1). This mainly comes from their differences in handling replacement misses as shown in Figure 7.6. DMM_WL program performs the best when the block size is set to 16. A 16×16 array block has a working size of 2KB, which uses 12% of the cache size. In DMM_LO, the whole cache size is used to group tasks. In the following, we will further present the effect of different blocking factors on cache performance. Both versions presented similar performance in terms of their compulsory misses and invalidations. This shows that the run-time system can minimize the data sharing overhead as well as the compiler-based technique for regular application DMM. The DMM_LO and DMM_WL had the nearly same number of replacement misses.

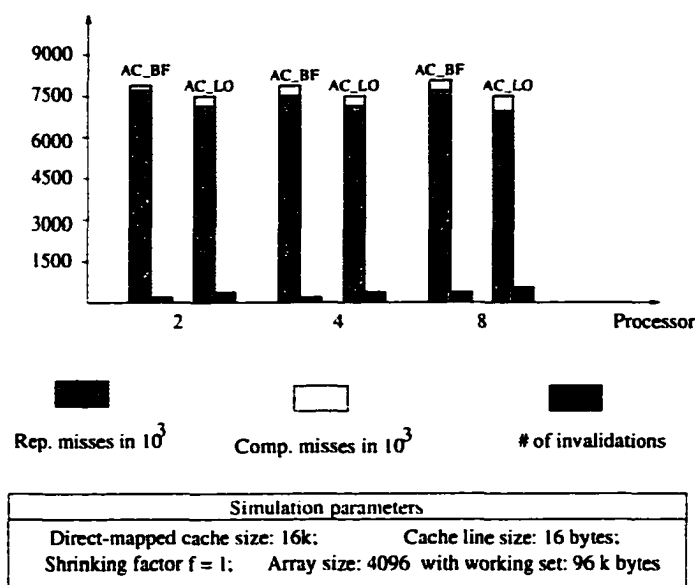


Figure 7.7: Cache performance comparison between AC_WL and AC_LO.

But, the former had a slightly larger number of replacement misses than the latter. This shows that the compiler-based technique can group tasks a little better than the run-time technique does. As the number of processors increases, the numbers of conflict misses and invalidations in both optimized versions increased, but the number of replacement misses decreased. The former is due to the increase in sharing degree among caches. The latter is due to the use of more caches. This is consistent with results in previous research work.

The AC application has a memory-access pattern which is not as regular as the DMM program. As shown the miss rates in Table 7.1, AC_LO, a locality optimized program of AC using the run-time technique, is shown to achieve slightly better cache performance than AC_BF, a well-tuned program. The number of both compulsory misses and replacement misses is improved by the run-time technique with respect to AC_BF. However, this improvement is mainly brought by the reduction in replacement misses because AC_LO causes a little more compulsory misses than AC_BF, as shown in Figure

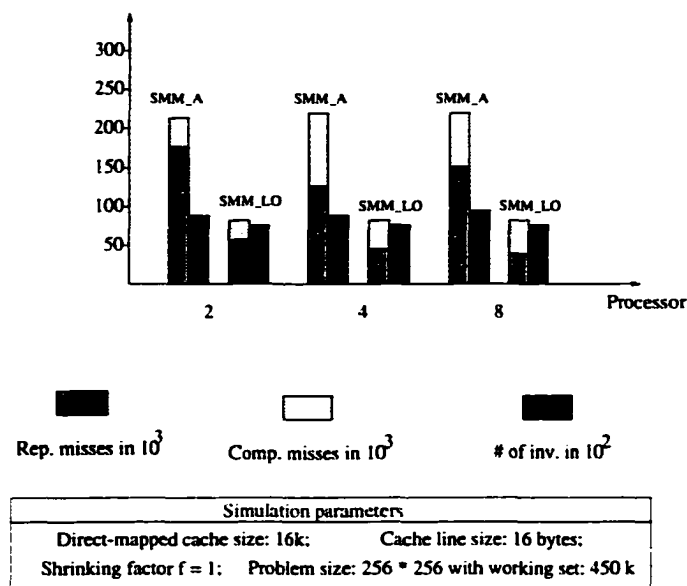


Figure 7.8: Cache performance comparison between SMM_A and SMM_LO.

7.7. But, AC_LO caused larger number of invalidations than AC_BF did. This shows that the static partitioning method used by AC_BF causes less data-sharing than that of the dynamic scheduling of the run-time technique. In addition, the distribution patterns of different types of misses in both programs did not show a significant change when the number of processors increased. This is different from the execution of the DMM program.

Regarding the application SMM, the run-time locality technique is shown to be very effective in reducing cache misses. The cache miss rate was reduced for more than 50% as shown in Table 7.1. This reduction is mainly from a significant reduction in replacement cache misses and a slight reduction in compulsory misses. Both SMM_LO and SMM_A present similar invalidation performance. These results show the great potential of optimizing the cache locality using run-time techniques for applications with dynamic memory-access patterns.

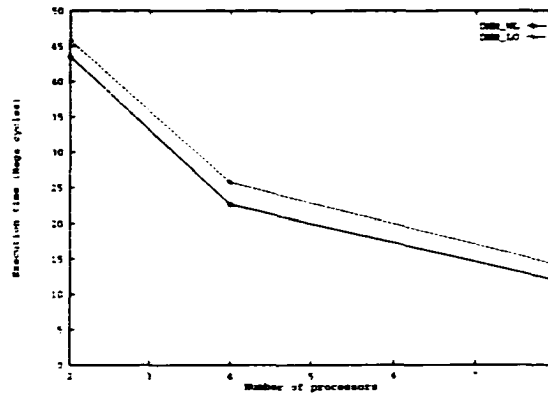


Figure 7.9: Execution time comparison between DMM_WL and DMM_LO.

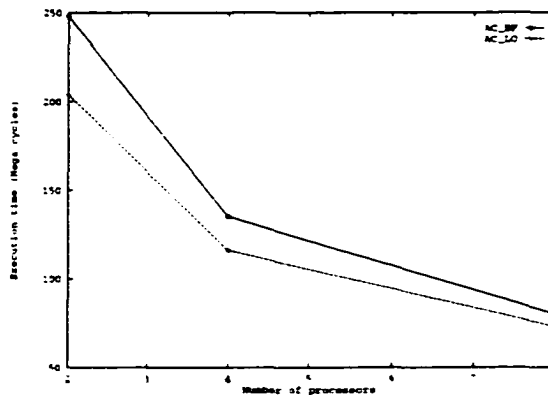


Figure 7.10: Execution time comparison between AC_BF and AC_LO.

B. Execution Performance and The Effects of Bus Traffic

The ultimate goal of the locality optimization is to reduce the execution time of an application. The execution performance of the locality-optimized programs on the run-time system is compared with their counterparts in Figures 7.9, 7.10, and 7.11. The performance differences between different parallel versions can be clarified by the differences in bus contention and load balance quality. The load balance measurements are presented in Table 7.2. The execution time is decomposed in Table 7.3 into three components: bus-retrying time, invalidation time, and data moving time.

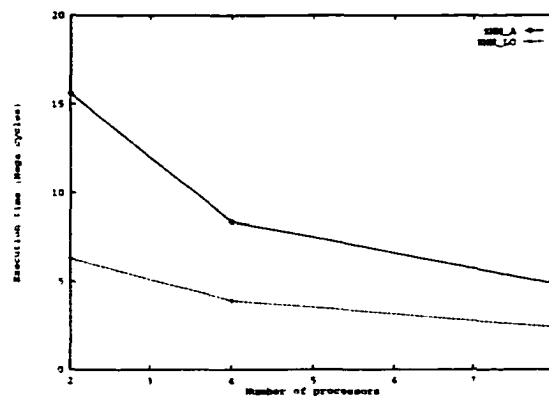


Figure 7.11: Execution time comparison between SMM_A and SMM_LO.

Regarding the DMM program, DMM_WL slightly outperformed DMM_LO. This is mainly because DMM_LO had worse load balance and longer data transferring time. Although the parallel iterations were perfectly partitioned among multiple processors in DMM_WL, slight load imbalance was also observed. This may be caused by bus retrying contention. Regarding the AC program, AC_LO outperformed AC_BF from 16% at 2 processors to 8% at 8 processors in terms of execution time. This improvement was also mainly contributed by certain reductions in bus-retry time and data transferring time. The AC_LO had worse load balance than AC_BF because AC_LO was trying to balance load based on pre-grouped tasks. However, this imbalance does not impact the overall performance significantly. This also shows that locality optimization is more important while load imbalance is not a major effect. Regarding the SMM program, SMM_LO performed much better than SMM_A by reducing 50% execution time using the run-time technique. The SMM_A still achieved a better load balance than SMM_LO. But, it caused higher bus-retrying contention and much longer data-transfer time due to a larger number of cache misses.

In order to study the effect of data transferring time on the overall execution time, we further differentiate the different types of data transfers. Table 7.4 presents the

Processors	Balance Coefficiency (%)					
	DMM application		AC application		SMM application	
	DMM_WL	DMM_LO	AC_BF	AC_LO	SMM_A	SMM_LO
2	1.10	1.3	0.001	0.11	0.03	0.18
4	1.48	1.47	0.11	0.61	0.24	0.34
8	1.90	1.95	0.20	0.12	1.9	3.0

Table 7.2: Load balance comparison where the load balance of each program is measured by the balance coefficiency defined in equation (7.1).

amount of data moved between different source-destination pairs. In all the programs, majority of data was moved from memory to caches and from caches to caches. There is only a small amount of data moved from caches to memory, which happened only when a dirty line was written back to memory before it was replaced. The amount of data that were moved between memory and caches decreased as the number of processors increased, because of the increase in the total capacity of available caches. But, the data communication traffic between caches was increased because more data was shared by more processors.

Compared with DMM_WL, DMM_LO caused more data movement by about 0.2MB from the memory to caches, about 0.02MB from caches to the memory, and 0.1MB from caches to caches. Regarding the AC program, AC_LO had a reduction of up to 13MB in the memory-to-cache transfer, a 0.7MB reduction in the cache-to-memory transfer, and a reduction of up to 6MB in the cache-to-cache transfer. For program SMM, SMM_LO achieved over 50% reduction in both the memory-to-cache transfer and the cache-to-memory transfer.

Dense matrix-matrix multiplication

Processors	DMM_WL			DMM_LO		
	bus-retry	bus-inv.	data-moving	bus-retry	bus-inv.	data-moving
2	0.29	0.0029	3.78	0.35	0.0030	5.4
4	0.60	0.0031	1.9	0.68	0.0034	3.4
8	0.88	0.0032	0.66	0.97	0.0039	1.3

Adjoint convolution

Processors	AC_BF			AC_LO		
	bus-retry	bus-inv.	data-moving	bus-retry	bus-inv.	data-moving
2	17.2	0.00005	79.7	15	0.006	66
4	31	0.00004	29	25	0.003	24
8	31.8	0.00003	9.6	29	0.002	9

Sparse matrix-matrix multiplication

Processors	SMM_A			SMM_LO		
	bus-retry	bus-inv.	data-moving	bus-retry	bus-inv.	data-moving
2	0.57	0.012	2.62	0.24	0.007	1.35
4	0.79	0.006	1.16	0.48	0.003	0.51
8	1.14	0.004	0.50	0.76	0.003	0.23

Table 7.3: Execution performance and bus traffic: All the timing results are given in 10^6 cycles. Load balance was measured by the ratio between execution-time derivation and the mean of the execution times of multiple processors. The locality optimized programs using our run-time approach use blocking factor 1.

Dense matrix-matrix multiplication

Processors	DMM_WL			DMM_LO		
	M2C	C2M	C2C	M2C	C2M	C2C
2	3.4	0.36	0.47	3.6	0.36	0.50
4	3.4	0.27	0.92	3.5	0.29	0.99
8	2.1	0.18	1.53	2.3	0.19	1.70

Adjoint convolution

Processors	AC_BF			AC_LO		
	M2C	C2M	C2C	M2C	C2M	C2C
2	84.6	1.0	40.1	71	0.33	37
4	51.6	1.0	74.7	46	0.34	73
8	22.6	0.99	104	20.7	0.36	98

Sparse matrix-matrix multiplication

Processors	SMM_A			SMM_LO		
	M2C	C2M	C2C	M2C	C2M	C2C
2	2.69	0.17	0.75	1.3	0.07	0.083
4	2.21	0.17	1.27	1.4	0.07	0.35
8	1.74	0.17	1.75	1.03	0.06	1.27

Table 7.4: Data movement traffic: M2C, C2M, and C2C, respectively, give the total amount of data in Mega bytes moved from memory to caches, from caches to memory, and from caches to caches. The locality optimized programs using our run-time technique use blocking factor 1.

applications	f	misses	miss-rate	comp.	rep.	inv.
DMM_LO	1	302	0.008	48	254	3201
	0.5	253	0.007	49	204	3501
	0.25	261	0.007	50	211	3224
	0.125	280	0.008	50	230	3240
AC_LO	1	7462	4.4	24	7438	412
	0.5	7476	4.4	24	7452	412
	0.25	7492	4.5	35	7457	412
	0.125	7538	4.6	45	7493	412
SMM_LO	1	85	0.011	33.6	52	8261
	0.5	88	0.012	44	44	8477
	0.25	107	0.012	45	62	8788
	0.125	107	0.013	45	62	8794

Table 7.5: Effects of varying blocking factor on cache performance (2 processors were chosen for DMM_LO, and 4 processors were chosen for both AC_LO and SMM_LO): **misses**, **comp.**, and **rep.**, respectively, give the numbers of misses, compulsory misses, and replacement misses in 10^3 ; and **inv.** gives the total number of invalidations.

C. Effects of interference

In our run-time technique, task grouping, which is controlled by factor f ($f \leq 1$), plays an important role in optimizing the memory performance of an application. The larger the f , the more number of tasks put in a group, possibly resulting in more cache interference. Cache interference refers to the overlapping degree of mapping addresses of data in a cache. Intuitively, grouping a small number of tasks to execute may reduce the overlapping degree of data access addresses in a cache. But, this also reduces the chance for tasks to reuse their data in a cache. How cache interference and data reuse affect the memory performance of an application is mainly determined by the memory-access pattern of an application. For our three applications, we investigate this effect by varying the blocking factor f .

Table 7.5 presents the changes of different cache misses as f is changed. For DMM_LO, the best cache performance is achieved with $f = 0.5$. No large variation was found for the numbers of compulsory misses and invalidations. For AC_LO, the cache performance became worse while f decreased. This phenomenon was also found for SMM_LO. Compared with AC_LO, SMM_LO was more sensitive to the change of f .

D. Run-time overhead

Run-time overhead is another important factor which affects the effectiveness of an run-time technique. In our proposed run-time technique, run-time overhead is mainly caused by task organization and task run-time scheduling. The task organization overhead is affected by the number of tasks created at run-time and the number of arrays accessed. The run-time scheduling overhead is affected by the imbalance in the initial task partition and in the run-time executions of multiple processors. Table 7.6 gives the percentage of the run-time overhead in the total execution time. For both DMM_LO and SMM_LO, the run-time overhead had a bigger influence on execution performance than AC_LO.

Applications	Processor		
	2	4	8
DMM_LO	4.1	3.2	2.8
AC_LO	0.23	0.3	0.20
SMM_LO	6	4.5	4

Table 7.6: Run-time overhead in percentage of total execution time.

This difference is mainly due to the difference in the computation granularities of tasks. The tasks in AC_LO had the largest computation granularity and the tasks in SMM_LO had the smallest computation granularity.

7.4.2 Measurements

Measurements on HP/CONVEX S-class

Measurement results of the different parallel versions on HP/CONVEX S-class are presented in Table 7.7. Regarding the DMM program, DMM_WL consistently performed a little bit better than DMM_LO. The better load balance in DMM_WL is a reason for this. For program AC, AC_LO performed much better than AC_BF on two processors. When more processors were applied, the execution times were close. But, AC_BF always balanced load better due to its perfect initial partition. But, the load imbalance occurred in the AL_LO was no larger than 1%. For SMM, SMM_LO had achieved a much better performance improvement over the SMM_A. This further confirms the effectiveness of the run-time technique in improving the performance of applications with dynamic memory-access patterns. However, SMM_A still achieved better load balance than SMM_LO. One reason for this is that SMM_LO used a locality preserved scheduling algorithm, which tried to keep the tasks in a group to execute together on a processor. This can increase

Application: Dense matrix multiplication

size	proc.	DMM_WL		DMM_LO		
		time	balance	time	overhead	balance
1024	2	11	0.0026	13	0.83	0.024
	4	5.7	0.0052	6.6	0.52	0.021
	8	3.0	0.0095	3.9	0.34	0.038
	16	1.8	0.010	2.2	0.24	0.040

Application: Adjoint convolution

size	proc.	AC_BF		AC_LO		
		time	balance	time	overhead	balance
400	2	180	0.0007	144	0.398	0.003
	4	102	0.0010	91	0.235	0.004
	8	65	0.0018	60	0.174	0.006
	16	39	0.0031	38	0.107	0.010

Application: Sparse matrix matrix multiplication

size	proc.	SMM_A		SMM_LO		
		time	balance	time	overhead	balance
1024	2	4.1	0.02	2.2	0.12	0.03
	4	2.5	0.03	1.3	0.11	0.05
	8	1.4	0.04	0.5	0.08	0.06
	16	0.8	0.06	0.5	0.01	0.06

Table 7.7: Execution time (in seconds) based comparison on HP/Convex S-class: Columns **time** and **overhead**, respectively, give total execution time and task organization overhead in second. Balance presents load balance measurements which is defined in equation (7.1). ($f = 1$).

Application	value of f			
	1	0.5	0.25	0.125
DMM_LO (N=1024)	6.6	6.1	5.8	5.8
AC_LO (N=400)	91	90	91	90
SMM_LO (N=1024)	1.3	1.3	1.4	1.5

Table 7.8: The effect of different values of f on execution time (in seconds) for DMM_LO and AC_LO on four processors of HP/Convex S-class.

data reuse in a cache. But, it also tends to cause more imbalance.

Table 7.7 also gives the run-time overhead of the task reorganization. Among all the applications, SMM_LO had the largest run-time overhead in term of the percentage in the total execution time, and AC_LO had the lowest. This is consistent with the simulation results. As mentioned before, this is mainly affected by the task granularity.

Regarding the effect of different values of f on performance, Table 7.8 presents the measurement results. For DMM_LO, the execution time decreased as f decreased, resulting in groups with a smaller number of tasks. The AC_LO is not sensitive to the change of f , which is consistent with our simulation results. The SMM_LO had longer execution time when a smaller f was used.

Measurements on HyperSPARC station-20

Table 7.9 gives the execution times of the parallel versions on HyperSPARC station-20, a much slower multiprocessor workstation than the S-class. The DMM_LO still achieved a close performance to DMM_WL, not worse than 9% in execution time. The run-time overhead in DMM_LO was about 10% of its execution time. For program AC, AC_LO outperformed AC_A for 8.5% in execution time reduction although it had worse load balance. Compared with SMM_A, SMM_LO reduced execution time up to 40%. These

Application: Dense matrix multiplication

size	proc.	DMM_WL		DMM_LO		
		time	balance	time	overhead	balance
1024	2	108	0.01	115	10	0.06
	4	57	0.02	63	7	0.03

Application: Adjoint convolution

size	proc.	AC_BF		AC_LO		
		time	balance	time	overhead	balance
256	2	763	0.002	698	0.67	0.003
	4	390	0.003	349	0.67	0.005

Application: Sparse matrix multiplication

size	proc.	SMM_A		SMM_LO		
		time	balance	time	overhead	balance
1024	2	37	0.012	23	2.0	0.035
	4	20	0.022	12	1.3	0.038

Table 7.9: Execution time (in seconds) based comparison on HyperSPARC station-20: Columns **time** and **overhead**, respectively, give total execution time and task organization overhead in second. **Balance** presents load balance measurements which is defined in equation (7.1). ($f = 1$).

Application	value of f			
	1	0.5	0.25	0.125
DMM_LO (N=1024)	63	64	58	59
AC_LO (N=256)	349	347	352	373
SMM_LO (N=1024)	12	13	14.6	14.2

Table 7.10: The effect of different values of f on execution time (in seconds) for DMM_LO and AC_LO on four processors of HyperSPARC station-20.

measurements are consistent with that on the S-class although the absolute performance results are different.

The effects of different values of f are presented in Table 7.10. The DMM_LO, AC_LO, and SMM_LO achieved the best performance respectively at $f = 0.25$, $f = 0.5$, and $f = 1$.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The locality of a program is affected by a wide range of performance factors. The design of efficient locality-optimization techniques relies on an insightful understanding of these performance factors. This dissertation models the locality optimization problem on uniprocessors and shows it to be an NP-complete problem. The locality optimization problem on uniprocessors is also a foundation for the optimization problem on multiprocessors. The dissertation provides an thorough analysis of the locality optimization problem on multiprocessors. As we point out, the non-deterministic factors in multiprocessor systems make the precise modeling of the locality optimization problem impossible. Based on the formal analysis of the locality optimization, a locality optimization technique should at least include the following three functionalities:

1. *Information acquisition*, which collects information on the cache-access pattern of a program. Here, the information on the data-access sequence of a program is essential for locality optimization. Higher precision in information acquisition is achieved at the cost of higher analysis complexity.

2. *Optimization*, which reorganizes the data layout and execution sequences of a program to maximize data reuse in caches and to minimize data sharing among caches.
3. *Integration*, which trades off locality with other performance factors to improve overall performance.

This dissertation proposes a run-time locality optimization technique, which targets applications with dynamic memory-access patterns. Previous experience in parallel computing has shown that real-world applications with irregular computational patterns and/or dynamic memory and data-dependence patterns are difficult to speedup, and are dominant among real-world applications [39]. The solutions to these applications rely on effective techniques to exploit the information of a program. The multi-dimensional internal structure proposed in this dissertation has been shown to be an effective way to integrate both static information and dynamic information. It allows the development of efficient run-time locality optimizations. Based on this internal structure, all the locality optimizations are implemented as a set of formal transformations that are represented by a compound hash function. The run-time overhead has been shown to be acceptable, which, in most cases, is not larger than 10% of the total execution time of a program. One important conclusion from this is that efficient run-time optimization and information acquisition techniques are able to be expressed in an integrated way in order to minimize the implementation overhead.

For the applications with dynamic memory-access patterns, we have shown that there is great potential for the run-time locality optimization technique to improve the performance. The data communication traffic on the interconnect network can be significantly reduced. Most importantly, this approach reduces the number of memory accesses to alleviate increasing demand on memory-bus bandwidth. In comparison with a regular application which was well-optimized by compiler-based techniques, we have shown that the run-time optimizations could perform competitively as well. Our run-

time system was implemented as a set of simple and portable library functions. It can be conveniently used by users on commercial SMPs. The run-time system is not aimed at replacing compiler-based techniques, but at complementing a compiler to optimize those applications that are beyond of its optimization capability. Furthermore, regarding an application with an irregular computational pattern but with a static memory-access pattern and a static data dependence pattern, the run-time technique can also achieve a little better performance than that of compiler-based optimizations. These results show the effectiveness of run-time techniques for a wide range of application patterns.

In the run-time system, task reordering, task partitioning and scheduling are three key optimization techniques. The effectiveness of the task reordering technique is shown by the reduction in the number of replacement misses and compulsory misses. Task partitioning and scheduling are effective as shown by the load-balance quality and the reduction of cache-to-cache data traffic. In the three optimized programs using the run-time technique, the deviation of execution times of multiple processors is no larger than 2% of the mean time. But, the cache-to-cache data traffic is still very large in some cases. This is because we consider load balance to have higher priority than locality in order to simplify the partitioning and scheduling procedure. So, determining an optimal tradeoff between load balance and locality is still an important open issue. Regarding the task scheduling problem, adaptive algorithms have been shown to be very effective in handling a wide range of load partitioning patterns.

The performance results were consistent across both simulation and measurement approaches.

8.2 Future work

Because memory access is becoming more and more expensive, effective techniques to improve the memory performance of applications are being aggressively pursued by cur-

rent computer industry. More techniques include run-time optimizations [2]. Although this dissertation has shown the potential and possibility of improving the performance of application using run-time locality optimizations, there are many aspects of this work that can be further extended. We point out some limits of our work and discuss possible solutions for addressing the limits.

8.2.1 Locality Modeling

In Section 3.3.2, we analyzed the locality optimization problem in uniprocessor systems and multiprocessor systems. In this dissertation, our analyses are aimed at understanding the effects of different factors in locality optimization to provide a guideline for the design of the run-time optimization technique. The development of a complete locality analysis theory and techniques is necessary.

Although the optimization problem in uniprocessors has been modeled in this dissertation, the model is based on the knowledge of the execution sequence of a program. In order to provide guidelines for sequential algorithm design, some techniques must be provided to conduct precise prediction on the execution sequence so that the locality of an algorithm can be predicted.

Regarding the locality optimization problem in multiprocessors, this dissertation only presents an approximate solution framework. One major difficulty in modeling the locality problem in multiprocessors comes from the nondeterministic interference among multiple processors, which is determined by the contention on the interconnection network and memory, and the dynamic features of the coherence protocol and the parallel execution. A possible solution to nondeterministic interference is to use a stochastic model, similar to those stochastic models used for modeling nondeterministic effects in task scheduling [49]. Because a parallel computing system is highly complex, it is not easy to develop a simple and effective stochastic model. Our previous research experi-

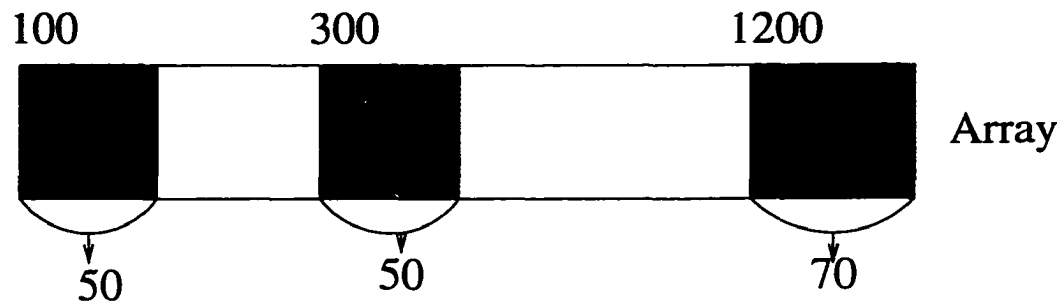


Figure 8.1: An noncontiguous access pattern of a task in an array where the array is linearly laid out in memory.

ence on performance modeling has shown that the practical applicability of a stochastic model is very restricted [87, 88, 89, 91, 92]. So, a more realistic approach is to develop a deterministic model to quantify the nondeterministic interference. One approach is given by reference [20], which proposes a deterministic performance model for parallel computing, called the LogP Model. One possible solution is to extend the LogP model with a consideration of the cache-access pattern of a program. Here, one challenging problem is how to integrate both program characteristics and cache architecture characteristics into the model effectively.

8.2.2 Information Estimation at Run-time

The effectiveness of a run-time locality optimization technique depends on how precisely the memory-access pattern of an application can be predicted. The multi-dimensional structure proposed in Section 4.2.2 can completely capture the whole memory-access space of an application. Regarding the prediction of the memory-access patterns of tasks, we only consider the starting physical addresses of the array-access regions of tasks. The simplicity of this prediction makes our formal transformations highly efficient. However, this prediction is obviously not precise.

An array-access region can only be captured by both its starting address and its

size, which can be expressed as an 2-tuple (`address`, `size`). In some parallel applications, a task may access several noncontiguous regions in an array. Figure 8.1 shows such a situation where the three noncontiguous regions in the array are accessed by a task. To capture this more complicated case, the array-access pattern of a task must be captured using multiple address-size pairs. For the example in Figure 8.1, the array-access pattern should be represented by three 2-tuples: (100, 50), (300, 50), and (1200,70). If there are M tasks and the average number of access-regions of each task in an array is T , the analysis on the relations among tasks will take $O(M^T)$ time because each contiguous region of a task should be analyzed with respect to all the contiguous regions of the other tasks (in contrast, the run-time optimization in our run-time system only has complexity $O(M)$). This would greatly increase run-time overhead.

How to trade off the prediction precision and the run-time overhead depends on whether the benefit of using more precise information can bring more performance improvement. Here, we need some way to estimate the performance improvement so that a proper tradeoff is made.

8.2.3 Run-time Optimizations

Task reordering and task partitioning are two major run-time optimizations conducted in our run-time system. Based on the collected information in the multi-dimensional memory-access space, tasks are grouped using a square space-shrinking method, which equivalently shrinks each dimension. However, when more information items on the array-access region are available, different dimensions should be shrunk by different factors. This problem should be studied together with the improvement of the prediction precision that is discussed in the last section.

Regarding the task partitioning, we have proposed a heuristic algorithm based on several discovered properties. This part can also be improved by investigating more

Type 4: The hardest application.

- * dynamic memory access pattern.
- * irregular computation pattern.
- * dynamic data-dependence.

```

double B[N], X[N], A[M];
int Row[N+1], Col[M];

trangular()
{
    int i, j;

    for (i=0; i<N; i++){
        X[i] = B[i];
        for (j=Row[i]; j<Row[i+1]; j++)
            X[i] -= A[j] * X[Col[j]];
    }
}

```

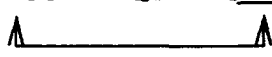


Figure 8.2: A Type 4 benchmark: Sparse Triangular Solver (STS). Here equation $B = A \times X$ is solved where A is a sparse lower triangular matrix with a dense representation.

properties. But, the use of more properties may incur more partitioning overhead. The effectiveness of using a more precise partitioning method is determined by two factors: its overhead and the data sharing degree of an application. How to make an optimal selection is really a difficult problem.

8.2.4 Programming Model Extension

Because we emphasize on the locality exploitation in this dissertation, the proposed programming model only takes into consideration the nested loops without data-dependence. However, many applications may not fit into this model due to loop-carried data dependence. In the classification of applications given in Section 4.3.2, the applications in Type 4 have irregular computational patterns, dynamic memory patterns, and dynamic

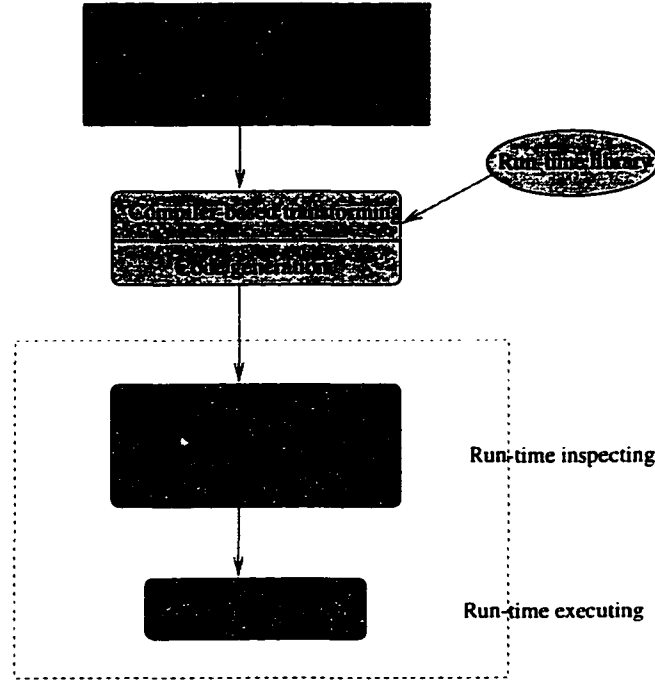


Figure 8.3: A framework of a run-time optimization system for all types of applications.

data-dependence patterns. This type of application is the most difficult for the locality optimization, because both the data-dependence and the locality optimization must be resolved at run-time. Here, we propose an approach to address this problem by combining our run-time technique with some existing research work on data dependence recognition.

Figure 8.2 gives a Type 4 benchmark, denoted the STS. Here, the k -th iteration of the loop with index variable i is considered as the k -th task, denoted t_k ($k = 0, 1, \dots, N$). The k -th task updates the k -th element of array A using several elements of array A which are indirectly determined by arrays Col and Row . Because arrays Col and Row could be input at run-time, the indirect accesses cannot be analyzed by a static compiler. So, the data dependence of the STS must be analyzed at run-time. For this type of applications, a run-time optimization system should integrate data-dependence

Type 4: The hardest application.

- * dynamic memory access pattern.
- * irregular computation pattern.
- * dynamic data-dependence.

```

double B[N], X[N], A[M];
int Row[N+1], Col[M];
int wave_front[N];
trangular()
{
    int i, j;
    int wf;
    for (i=0; i<N; i++){
        wf = 0;
        for (j=Row[i]; j<Row[i+1]; j++)
            if (wf < wave_front[Col[j]])
                wf = wave_front[Col[j]];
        wave_front[i] = wf + 1;
    }
}

```

Figure 8.4: A generated wave-front analysis program of the STS [64].

analysis and locality optimization. Based on this and the run-time system framework 4.1, an integrated system framework for dependence analysis and locality optimization at run-time is described in Figure 8.3. For a given program, the compiler is responsible for generating analysis program to analyze data dependence and for inserting run-time library functions to optimize the locality. At run-time, the data-dependence of a program is first analyzed, then the locality is optimized based on the analyzed data-dependence.

The run-time analysis techniques for data dependence have been studied previously. A wave-front based analysis technique was proposed by Saltz and Mirchandaney [64]. Here, we describe how to incorporate their technique with our locality optimization technique to deal with the most difficult type of applications, such as STS. For any given set of tasks, their data-dependence can be expressed as a dataflow graph where each node represents a task and each edge represents that the sink node needs the output of the

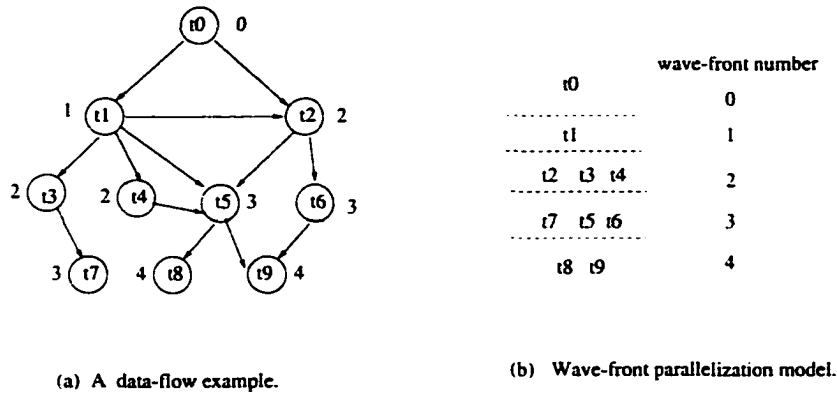


Figure 8.5: An example for the wave-front analysis.

source node. All the nodes without input edges are called starters. An example dataflow graph is shown in Figure 8.5(a) where only t_0 is a starter. The wave-front number of each node is defined as the length of the longest path from the node to the starters. For the STS, a compiler will generate a wave-front analysis procedure, which is presented in Figure 8.4, to calculate the wave-front numbers of each iteration of the loop with index i . Based on wave-front numbers of tasks, tasks are classified into different wave-front groups where all tasks with the same wave-front number are in a group. For the example in Figure 8.5(a), tasks are classified into five wave-front groups. Associated with the wave-front groups, the following two properties hold:

- All the tasks in a wave-front group are data independent, i.e., there are no edges from one task to another in the group. So, the locality of the tasks in a wave-front can be optimized using our run-time technique.
- A task in a wave-front group only needs data from the tasks with smaller wave-front numbers. This determines the execution order of wave-front groups.

To integrate the wave-front analysis technique in our run-time system, we can simply split each task bin as a set of sub-bins where each sub-bin corresponds to a different wave-front number. Abstractly, the original multi-dimensional space is augmented with

one more dimension, the wave-front dimension. So, the mapping of a task into the new multi-dimensional space is finished by the combination of the original compound hash function and the wave-front number of the task.

Bibliography

- [1] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, pp. 66-76, December 1996.
- [2] S. V. Adve, et al, "Changing Interaction of Compiler and Architecture, " *IEEE Computer*, pp. 51-58, December 1997.
- [3] A. Agarwal, D. Kranz, and V. Natarajan, "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared Memory Multiprocessors, " *IEEE Transactions on Parallel and Distributed Systems*, No. 9, Vol. 6, pp. 943-962, Sept. 1995.
- [4] A. Agarwal and S. D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct Mapped Caches, " *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 179-190, May 1993.
- [5] J. M. Anderson, S. P. Amarasinghe and M. S. Lam, "Data and Computation Transformations for Multiprocessors, " *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 166-178, July 1995.
- [6] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines, " *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 112-125, June 1993.

- [7] G. Astfalk and T. Brewer, "An Overview of the HP/Convex Exemplar Hardware, " Hewlett-Packard Company, Convex Technology Center, March 1997.
- [8] U. Banerjee, "Unimodular Transformations of double loops, " *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pp. 192–219, August 1990.
- [9] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches, " *Proceedings of Sixth International Conference on Architectural Support for Programming Language and Operating*, pp. 158–170, Oct. 1994.
- [10] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System, " *IEEE Computer*, 23(5), pp. 35–43, May 1990.
- [11] W. J. Bolosky, M. L. Sott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "NUMA policies and their relation to memory architecture, " *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 212–221, April 1991.
- [12] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum and M. S. Lam, "Compiler-Directed Page Coloring for Multiprocessors, " *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems, (ASPLOS VII)*, pp. 244–255, Oct 1996.
- [13] Doug Burger, J. R. Goodman, Alain Kagi, "Limited Bandwidth to Affect Processor Design, " *IEEE Micro*, pp. 55–62, November/December 1997.
- [14] M. Cekanov, et. al. , "SPARCcenter 2000: Multiprocessing for the 90's, " *Proceedings of IEEE COMPCON*, pp. 345–353, Feb. 1993.

- [15] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and Page Migration for Multiprocessor Compute Servers, " *Proceedings of Sixth International Conference on Architectural Support for Programming Language and Operating*, pp. 12–24, Oct. 1994.
- [16] R. Chandra and A. Gupta and J. Hennessy, "Data Locality and Local Balancing in COOL, " *Proceedings of Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pp. 249–259, May 1993.
- [17] S. Coleman and K. S. Mckinley, "Tile Size Selection Using Cache Organization and Data Layout, " *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp. 279–290, June 1995.
- [18] CONVEX Computer Corporation, *CONVEX Exemplar Architecture*, Second Edition, Document #710-004730-001, November, 1994.
- [19] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, INC., 1997.
- [20] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. V. Eicken, "LogP: Towards a Realistic Model of Parallel Computation, " *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. , May 1993.
- [21] E. V. D. Deijl, O. Temam, E. Granston, and G. Kanbier, " The Cache Visualization Tool (CVT), " *IEEE Computer*, pp. 71–78, July 1997.
- [22] J. Ferrante, V. Sarkar, and W. Thrash, "On estimating and enhancing cache effectiveness, " *Proceedings of 4th International Workshop in Lanuages and Compilers for Parallel Computing*, pp. 587–616, 1991.

- [23] C. Fricker, O. Temam, W. Jalby, "Influence of Cross-interferences on Blocked Loops: A Case Study with Matrix-Vector Multiply, " *ACM Transactions on Programming Languages and Systems*, 17(4), pp. 561–575, July 1995.
- [24] M. Galles and E. Williams, "Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor, " *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, Volum I: Architecture, pp. 134–143, Jan. 1994.
- [25] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation, " *Journal of Parallel and Distributed Computing*, 5(5), pp. 587–616, Oct. 1988.
- [26] A. J. Goldberg and J. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL, " *Proceedings of Supercomputing'91*, pp. 481–490, Nov. 1991.
- [27] A. Gupta, M. Martonosi, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs, " *Performance Evaluation Review*, 20(1), pp. 1–12, June 1992.
- [28] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods of the Performance of Parallel Applications, " *Proceedings of SIGMETRICS'91*, pp. 120–132, May 1991.
- [29] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient Procedure Mapping Using Cache Line Coloring, " *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 171–182, June 1997.
- [30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, INC., 1996.

- [31] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches, " *IEEE Transactions on Computers*, 38(12), pp. 1612–1630, Dec. 1989.
- [32] S. E. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a practical and robust method for scheduling parallel loops," *Communications of the ACM*, Vol. 35. No. 8, pp. 90-101, 1992.
- [33] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance With an Optimizing Compiler, " *Proceedings of 16th Annual International Conference on Computer Architecture*, pp. 242–251, 1989.
- [34] T. E. Jeremiassen and S. J. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformations, " *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 179–188, July 1995.
- [35] C. Jin, Y. Yan and X. Zhang, "An Adaptive Loop Scheduling Algorithm on Shared-Memory Systems, " *Proceedings of the Eighth IEEE Symposium of Parallel and Distributed Processing (SPDP'96)*, IEEE Computer Society Press, October 1996, pp. 250–257.
- [36] T. L. Johnson and W. W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis, " *Proceedings of ISCA '97*, pp. 315–326, July 1997.
- [37] N.P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, " *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364–373, May 1990.
- [38] Kendall Square Research, *KSR-1 Technology Background*, 1992

- [39] K. Kennedy and K. S. Mckinley, "Optimizing for Parallelism and Data Locality, " *Proceedings of the 1992 ACM International Conference on Supercomputing*, pp. 323–334, July 1992.
- [40] R. Kessler and M. D. Hill, "Page Placement Algorithms for Large Real-indexed Caches, " *ACM Transactions on Computer Systems*, 10(4), pp. 338–359, Nov. 1992.
- [41] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms, " *Proceedings of fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 94–105, April 1991.
- [42] R. P. LaRowe and C. S. Ellis, "Experimental Comparison of Memory Mangement Policies for NUMA Multiprocessors, " *ACM Transactions on Computer Systems*, 9(4), pp. 319–363, Nov. 1991.
- [43] A. R. Lebeck and D. A. Wood, "Cache Profiling and SPEC Benchmarks: A Case Study, " *IEEE Computer*, pp. 15–26, Oct. 1994.
- [44] D. Lenoski et al. , "The DASH Prototype:Logic Overhead and Performance, " *IEEE Transactions on Parallel and Distributed Systems*, Vol.4, No.1, 1993, pp. 41-61.
- [45] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and Loop Scheduling on NUMA Multiprocessors, " *International Conference on Parallel Processing*, Vol. II, pp. 140–144, 1993.
- [46] W. Li and K. Pingali, "Access Normalization: Loop Restructuring for NUMA Compilers, " *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 285–295, 1992.

- [47] J. Liu, V. A. Saletore, and T. G. Lewis, "Safe Self-scheduling: a parallel loop scheduling scheme for shared-memory multiprocessors", *International Journal of Parallel Programming*, Vol. 22, No. 6, pp. 589-616, 1994.
- [48] D. B. Loveman, "High Performance Fortran, " *IEEE Parallel & Distributed Technology*, 1(1), pp. 25-42, Feb. 1993.
- [49] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs, " *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1996, pp. 200-212.
- [50] N. Manjikian and T. S. Abdelrahman, "Fusion of Loops for Parallelism and Locality", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 2, pp. 193-209, Feb. 1997.
- [51] E. P. Markatos and T. J. Leblanc, "Using Processor Affinity in Loop Scheduling Scheme on Shared-Memory Multiprocessors, " *IEEE Transactions on Parallel and Distributed Systems*, 5(4), pp. 379-400, April 1994.
- [52] S. McFarling, "Program Optimization for Instruction Caches, " *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183-191, April 1989.
- [53] S. McFarling, "Cache Replacement with Dynamic Exclusion, " *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 191-200, May 1992.
- [54] K. S. McKinley, S. Carr, and C. W. Tseng, "Improving Data Locality with Loop Transformations, " *ACM Transactions on Programming Languages and Systems*, 18(4), pp. 424-453, July 1996.

- [55] K. S. Mckinley and O. Teman, "A Quantitative Analysis of Loop Nest Locality, " *The Proceedings of Seventh International Conference on Architectural Support for Programming languages and Operating Systems*, pp. 94–104, 1996.
- [56] J. C. Mogul and A. Gorg, "The Effect of Context Switches on Cache Performance, " *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 75–84, April 1991.
- [57] L. M. Ni and C. E. Wu, "Design tradeoffs for process scheduling in shared memory multiprocessor System", *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, pp. 327–334, 1989.
- [58] N. J. Nilsson, Ed., *Principles of Artificial Intelligence*, Tioga Publishing Co., 1980.
- [59] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning, " *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.
- [60] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread Scheduling for Cache Locality, " *The Proceedings of Seventh International Conference on Architectural Support for Programming languages and Operating Systems*, pp. 60–71. Oct. 1996.
- [61] C. Polychronopoulos and D. Kuck, "Guided self-scheduling: a practical self-scheduling scheme for parallel supercomputers", *IEEE Transactions on Computers*, Vol. C-36, No. 12, pp. 1425–1439, 1987.
- [62] T. H. Romer, D. Lee, B. N. Bershad and J. B. Chen, "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware, " *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 255–266, Nov. 1994.

- [63] S. Sahni, "General Techniques for Combinational Approximation," *Operation Research*, 25(6), pp. 920–936, 1977.
- [64] J. H. Saltz and R. Mirchandaney, "Run-Time Parallelization and Scheduling of Loops," *IEEE Transactions on Computers*, Vol. 40, No. 5, pp. 603–612, May 1991.
- [65] M. S. Squillante and E. D. Lazowska, "Using Processor-Cache Affinity in Shared-Memory Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, 4(2): 131–143, February 1993.
- [66] M. B. Steinman, G. J. Harris, A. Koccev, V. C. Lamere, and R. D. Pannell, "The AlphaServer 4100 Cached Processor Module Architecture and Design," *Digital Technical Journal*, 8(4), pp. 21–37, 1996,
- [67] P. Stenstrom, E. Hagersten, D. J. Lilja, M. Martonosi and M Venugopal, "Trends in Shared Memory Multiprocessing," *IEEE Computer*, pp. 44–50, December 1997.
- [68] S. Subramaniam and D. L. Eager, "Affinity scheduling of unbalanced workloads", *Supercomputing'94*, pp. 214–226, 1994.
- [69] P. Tang and P. C. Yew, "Processor self-scheduling for multiple nested parallel loops", *Proceedings of 1986 International Conference on Parallel Processing*, pp. 528–535, 1986.
- [70] J. Tartalia and V. Milutinovic, "Classifying Software-Based Cache Coherence Solutions," *IEEE Software*, pp. 90–101, May/June 1997.
- [71] O. Temam, C. Fricker, and W. Jalby, "Impact of Cache Interferences on Usual Numerical Dense Loop Nests," *Proceedings of The IEEE*, Vol. 81, No. 8, pp. 1103–1115, August 1993.

- [72] M. Tomasevic and V. Milutinovic, "A Survey of Hardware Solutions for Maintenance of Cache Consistency in Shared Memory Multiprocessor Systems, " *IEEE Micro* (Part #1, pp. 52–59, October 1994.) and (Part #2, pp. 61–66, December 1994).
- [73] J. Torrellas, M. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches, " *IEEE Transactions on Computers*, 43(6), pp. 651–663. June 1994.
- [74] J. Torrellas, A. Tucker, and A. Gupta, "Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors, " *Proceedings of SIGMETRICS'93*, pp. 272–274, 1993.
- [75] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads, " *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pp. 360–369, January 1995.
- [76] A. Tucker and A. Gupta, "Processor Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessor, " *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 159–166, 1989.
- [77] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, pp. 87-98, 1993.
- [78] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors, " *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 26–40, Oct. 1991.

- [79] J. E. Veenstra and R. J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," *Proceedings of MASCOTS'94*, pp. 201–207, Jan. 1994.
- [80] B. Verghese, S. Devine, A. Gupta and M. Rosenblum, "Operating System Support for Improving Locality on CC-NUMA Compute Servers," *The Proceedings of Seventh International Conference on Architectural Support for Programming languages and Operating Systems*, pp. 279–289, Oct. 1996.
- [81] M. Wolfe, *High Performance Compilers For Parallel Computing*, Addison-Wesley Publishing Company, Inc., 1996.
- [82] M. E. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.
- [83] Y. Yan, C. M. Jin, and X. Zhang, "Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, 8(1), pp. 70–81, Jan. 1997.
- [84] Y. Yan and Xiaodong Zhang, "A Memory-layout Oriented Run-Time Approach for Locality Optimization of Parallel Loops," *IEEE International Conference on Parallel Processing*, August, 1998.
- [85] Y. Yan, X. Zhang, and M. Qian, "Software and Visualization Support to Performance Evaluation of Parallel Computing Scalability," *IEEE Transactions on Software Engineering*, Vol. 23, No. 1, Jan. 1997, pp. 4–15.
- [86] C. Zhang, X. Zhang, and Y. Yan, "Two Fast and High-Associativity Cache Schemes," *IEEE Micro*, pp. 40–49, Sept./Oct. 1997.

- [87] X. Zhang and Y. Yan, "Latency Analysis of CC-NUMA and CC-COMA Hierarchical Rings, " *Proceedings of International Conference on Parallel Processing (ICPP'94)*. CRC Press, Vol. I, August, 1994, pp 174–181.
- [88] X. Zhang and Y. Yan, "Modeling Data Migration on CC-NUMA and CC-COMA Ring Architectures, " *Proceedings of MASCOTS 94, international Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society Press, January, 1994, pp. 159–164.
- [89] X. Zhang and Y. Yan, " Comparative Modeling and Evaluation of CC-NUMA and CC-COMA on Hierarchical Ring Architectures, " *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 12, 1995, pp. 1316–1331.
- [90] X. Zhang, Y. Yan, and R. Castaneda, "Evaluating and Designing Software Mutual Exclusion Algorithms on Shared Memory Multiprocessors, " *IEEE Parallel & Distributed Technology*, Spring Issue, 1996, pp. 25–42.
- [91] X. Zhang, Y. Yan, and R. Castañeda, "Comparative Performance Analysis and Evaluation of Hot Spots on MIN-Based and HR-based Shared-memory Architectures. " *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 8. Aug. 1995. pp. 872–886.
- [92] X. Zhang, Y. Yan, and R. Castaneda, "Modeling and Measuring Hot Spots on MIN-Based and HR-based Shared-memory Architectures, " *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society Press, 1993, pp. 176–183.
- [93] X. Zhang, Y. Yan, and K. He, "Latency Metric: An Experimental Method for Measuring and Evaluating Program and Architecture Scalability, " *Journal of Parallel and Distributed Computing*, Vol. 20, No. 9, 1994, pp. 392–410.

-
- [94] X. Zhang, Y. Yan, and K. He, "Evaluation and Measurement of Multiprocessor Latency Patterns, " *Proceedings of International Parallel Processing Symposium (IPPS'94)*, IEEE Computer Society Press, April, 1994, pp. 845–852.

VITA

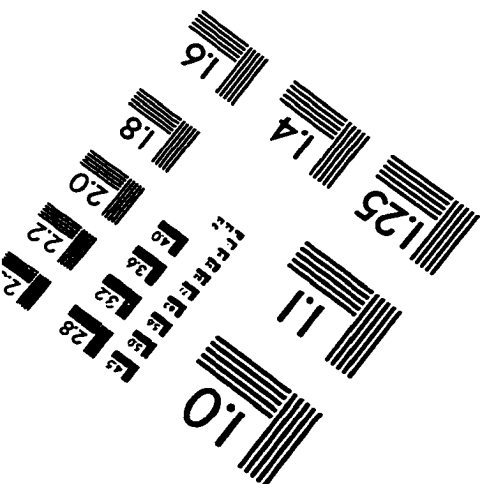
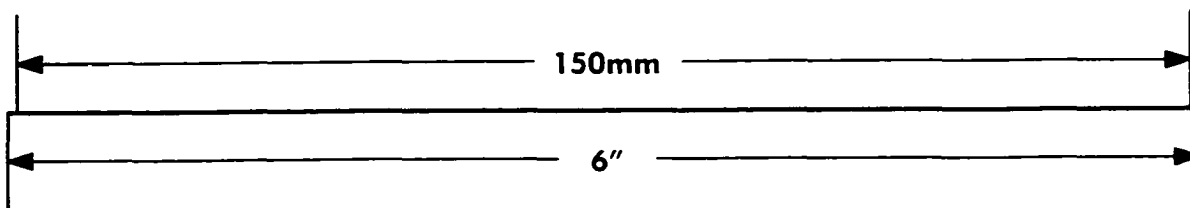
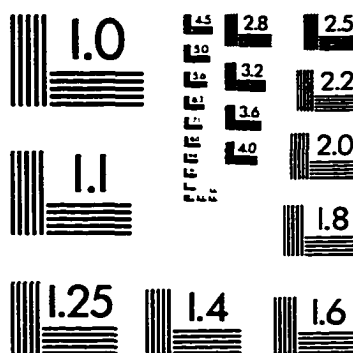
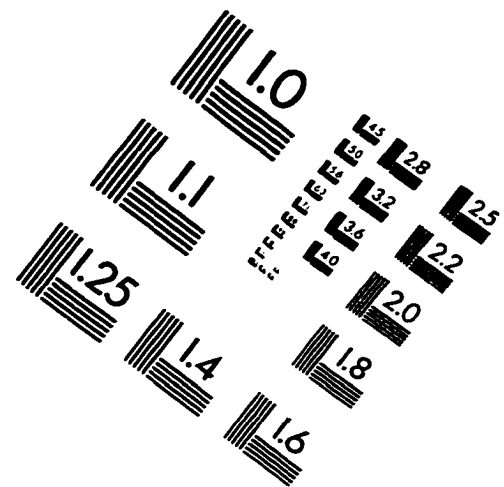
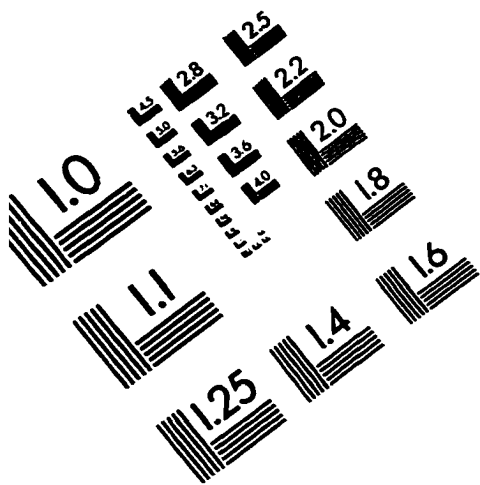
The author was born in Shishou county, Hubei province in P. R. China on November 20, 1963. He received a Bachelor Degree and his first Master Degree in Computer Science from Huazhong University of Science and Technology (HUST) in P. R. China, respectively in 1984 and in 1987. He received his second Master Degree in Computer Science from the University of Texas at San Antonio in 1997.

From May 1987 to December 1990, he worked as an assistant professor in the Department of Computer Science in HUST. From January 1991 to March 1993, he was an associate professor there. During this period of time, his research focused on distributed systems, distributed algorithms, parallel compilers. Meanwhile, he taught Parallel Processing, Distributed Systems, Artificial Intelligence, and System Programming.

In March 1993, invited by Dr. Xiaodong Zhang, he came to U. S. A as a visiting scholar to join the High Performance Computing and Software Lab in the University of Texas at San Antonio (UTSA). His research concentrated on metrics and visualization environments for evaluating the scalability of numerical applications and on fast read/write based software synchronization algorithms for shared memory systems

In September 1995, he became a Ph.D. student in the Division of Computer Science at UTSA, and became a Ph.D. candidate in December 1995. To continue his Ph.D research work, he followed his advisor to join the Ph.D program of the Department of Computer Science in the College of William and Mary in August 1997. In the past three years of his Ph.D. studies, his research emphasized on the performance evaluation of shared memory systems, metrics for network computing, new cache architectures and run-time approaches for improving the memory performance of parallel applications. This thesis reflects a part of his latest research work. So far, he has published about 50 technical papers in Journals and Conferences.

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

